

Python3 入門

Kivy による GUI アプリケーション開発,
サウンド入出力,
ウェブスクレイピング

第 3.3.16 版

Copyright © 2017-2025, Katsunori Nakamura

中村勝則

2025 年 9 月 16 日

免責事項

本書の内容は参考資料であり、掲載したプログラムリストは全て試作品である。本書の使用に伴って発生した不利益、損害の一切の責任を筆者ならびに IDEJ 出版は負わない。

目次

1	はじめに	1
1.1	Python でできること	1
1.2	本書の内容	1
1.3	本書の読み方	1
1.4	処理系の導入（インストール）と起動の方法	2
1.4.1	Python 処理系のディストリビューション（配布形態）	2
1.4.2	Python 処理系の起動	2
1.4.3	対話モード：REPL	3
1.4.3.1	Python 言語処理系の終了	3
1.4.3.2	ヒストリ	3
1.4.3.3	直前の値	3
1.4.3.4	REPL の機能に関すること	4
1.5	本書で取り扱う GUI ライブラリ	4
1.6	表記に関する注意事項	4
1.7	Python に関する詳しい情報	4
2	Python の基礎	5
2.1	スクリプトの実行	5
2.1.1	プログラム中に記述するコメント	6
2.1.2	プログラムのインデント	7
2.2	文を分割して／連結して記述する方法	7
2.3	文の記述の省略	8
2.4	変数とデータの型	8
2.4.1	日本語の変数名の使用	9
2.4.2	変数名に関する注意	9
2.4.3	イコール「=」の連鎖（Multiple Assignment）	10
2.4.4	変数の解放（廃棄）	10
2.4.5	変数が値を保持する仕組みの概観	11
2.4.5.1	識別値に関する注意事項	12
2.4.6	数値	12
2.4.6.1	整数： int 型	13
2.4.6.2	巨大な整数値を扱う際の注意	13
2.4.6.3	浮動小数点数： float 型	14
2.4.6.4	アンダースコア「_」を含む数値リテラルの記述	16
2.4.6.5	基数の指定： 2 進数, 8 進数, 16 進数	16
2.4.6.6	複素数： complex 型	16
2.4.6.7	各種の数の型と数学的階層	18
2.4.6.8	異なる型の数値同士の算術演算と型昇格規則	18
2.4.6.9	最大値, 最小値	18
2.4.6.10	浮動小数点数の誤差と丸め	18
2.4.6.11	数学関数	19
2.4.6.12	特殊な値： inf, nan	22
2.4.6.13	多倍長精度の浮動小数点数の扱い	25
2.4.6.14	分数の扱い	28
2.4.6.15	除算における商と剰余に関する事柄	29
2.4.6.16	正確な 10 進演算	31

2.4.6.17	乱数の生成	35
2.4.7	文字列	38
2.4.7.1	エスケープシーケンス	38
2.4.7.2	raw 文字列 (raw string)	39
2.4.7.3	複数行に渡る文字列	39
2.4.7.4	文字列の長さ (文字数)	40
2.4.7.5	文字列リテラルの接続	40
2.4.7.6	文字列の分解と合成	40
2.4.7.7	文字列の置換	42
2.4.7.8	文字の置換	43
2.4.7.9	英字, 数字の判定	43
2.4.7.10	大文字／小文字の変換と判定	44
2.4.7.11	文字列の含有検査	44
2.4.7.12	文字列の検索	45
2.4.7.13	両端の文字の除去	46
2.4.7.14	文字コード, 文字の種別に関すること	47
2.4.7.15	文字列のデータサイズについて	49
2.4.8	真理値 (bool 型)	50
2.4.9	ヌルオブジェクト: None	50
2.4.10	式や値の記述の省略	51
2.4.11	型の変換	51
2.4.11.1	各種の値の文字列への変換 (str, repr)	51
2.4.12	型の検査	52
2.4.12.1	float の値が整数値かどうかを検査する方法	53
2.4.13	基数の変換	53
2.4.13.1	n 進数 → 10 進数	53
2.4.13.2	10 進数 → n 進数	54
2.4.14	ビット演算	54
2.4.14.1	ビット演算の累算代入	55
2.4.14.2	整数値のビット長を求める方法	55
2.4.15	バイト列	56
2.5	データ構造	57
2.5.1	リスト	57
2.5.1.1	空リスト	57
2.5.1.2	リストの要素へのアクセス	57
2.5.1.3	リストの編集	58
2.5.1.4	リストによるスタック, キューの実現: pop メソッド	61
2.5.1.5	リストに対する検査	62
2.5.1.6	例外処理	63
2.5.1.7	要素の個数のカウント	64
2.5.1.8	要素の合計: sum 関数	65
2.5.1.9	要素の整列 (1): sort メソッド	65
2.5.1.10	要素の整列 (2): sorted 関数	66
2.5.1.11	要素の順序の反転	66
2.5.1.12	リストの複製	66
2.5.1.13	リストか否かの判定	68
2.5.1.14	リストの内部に他のリストを展開する方法	68
2.5.2	タプル	68

2.5.2.1	リストとタプルの違い	69
2.5.2.2	括弧の表記が省略できるケース	70
2.5.2.3	特殊なタプル	70
2.5.2.4	タプルの要素を整列 (ソート) する方法	70
2.5.2.5	タプルか否かの判定	71
2.5.3	セット	71
2.5.3.1	セットの要素となるオブジェクト	71
2.5.3.2	セットの生成	72
2.5.3.3	セットに対する各種の操作	73
2.5.3.4	集合論の操作	74
2.5.3.5	セットか否かの判定	75
2.5.3.6	frozenset	75
2.5.4	辞書型	76
2.5.4.1	空の辞書の作成	76
2.5.4.2	エントリの削除	76
2.5.4.3	エントリの存在検査	77
2.5.4.4	get メソッドによる辞書へのアクセス	77
2.5.4.5	全てのエントリの削除	77
2.5.4.6	辞書に対する pop	78
2.5.4.7	辞書の更新	78
2.5.4.8	辞書の結合	79
2.5.4.9	辞書を他の辞書の内部に展開する方法	79
2.5.4.10	キーや値の列を取り出す方法	79
2.5.4.11	エントリの数を調べる方法	80
2.5.4.12	リストやタプルから辞書を生成する方法	80
2.5.4.13	辞書の全エントリを列として取り出す方法	81
2.5.4.14	辞書の複製	81
2.5.4.15	エントリの順序	81
2.5.4.16	辞書の整列	82
2.5.4.17	辞書の同値性	82
2.5.4.18	辞書か否かの判定	82
2.5.5	添字 (スライス) の高度な応用	83
2.5.5.1	添字の値の省略	83
2.5.5.2	スライスオブジェクト	83
2.5.5.3	逆順の要素指定	83
2.5.5.4	不連続な部分の取り出し	84
2.5.6	データ構造の変換	84
2.5.7	データ構造の要素を他のデータ構造の中に展開する方法	85
2.5.8	データ構造に沿った値の割当て (分割代入)	85
2.5.8.1	応用例 1: 変数の値の交換	86
2.5.8.2	応用例 2: 高度な代入処理	86
2.5.8.3	データ構造の選択的な部分抽出	87
2.5.9	データ構造のシャッフル, ランダムサンプリング	87
2.5.10	データ構造へのアクセスの速度について	88
2.5.11	累算代入によるデータ構造の拡張に関する注意点	88
2.6	制御構造	89
2.6.1	繰り返し (1): for	89
2.6.1.1	イテラブルなオブジェクト	89

2.6.1.2	「スイート」の概念	89
2.6.1.3	スイートが 1 行の場合の書き方	90
2.6.1.4	range 関数, range オブジェクト	90
2.6.1.5	辞書の for 文への応用	91
2.6.1.6	for 文における else	91
2.6.1.7	for を使ったデータ構造の生成 (要素の内包表記)	92
2.6.1.8	イテレータ	92
2.6.1.9	分割代入を用いた for 文	94
2.6.1.10	zip 関数と zip オブジェクト	94
2.6.1.11	enumerate によるインデックス情報の付与	96
2.6.2	繰り返し (2): while	96
2.6.3	繰り返しの中断とスキップ	97
2.6.4	条件分岐	97
2.6.4.1	条件式	98
2.6.4.2	比較演算子の連鎖	98
2.6.4.3	データ構造の比較	99
2.6.4.4	各種の「空」値に関する条件判定	100
2.6.4.5	is 演算子による比較	101
2.6.4.6	値の型の判定	102
2.7	入出力	103
2.7.1	標準出力	103
2.7.1.1	出力データの書式設定	103
2.7.1.2	sys モジュールによる標準出力の扱い	106
2.7.2	標準入力	107
2.7.2.1	input 関数による入力の取得	107
2.7.2.2	sys モジュールによる標準入力の扱い	108
2.7.3	ファイルからの入力	108
2.7.3.1	ファイル, ディレクトリのパスについて	109
2.7.3.2	扱うファイルのエンコーディング, 改行コードの指定	110
2.7.3.3	テキストファイルとバイナリファイルについて	112
2.7.3.4	バイト列の扱い	112
2.7.3.5	バイト列のコード体系を調べる方法	114
2.7.3.6	指定したバイト数だけ読み込む方法	114
2.7.3.7	ファイルの内容を一度で読み込む方法	114
2.7.3.8	ファイルをイテラブルとして読み込む方法	115
2.7.3.9	readlines メソッドによるテキストファイルの読み込み	115
2.7.3.10	データを読み込む際のファイル中の位置について	116
2.7.4	ファイルへの出力	116
2.7.4.1	print 関数によるファイルへの出力	116
2.7.4.2	writelines メソッドによる出力	117
2.7.4.3	出力バッファの書き出し	117
2.7.5	標準エラー出力	117
2.7.6	標準入出力でバイナリデータを扱う方法	119
2.7.7	パス (ファイル, ディレクトリ) の扱い: その 1 - os モジュール	120
2.7.7.1	カレントディレクトリに関する操作	120
2.7.7.2	ホームディレクトリの取得	120
2.7.7.3	ディレクトリ内容の一覧 (1)	120
2.7.7.4	ディレクトリ内容の一覧 (2)	121

2.7.7.5	ファイルのサイズの取得	121
2.7.7.6	ファイル、ディレクトリの検査	121
2.7.7.7	ファイル、ディレクトリの削除	122
2.7.7.8	実行中のスクリプトに関する情報	122
2.7.7.9	パスの表現に関すること	122
2.7.8	パス（ファイル、ディレクトリ）の扱い：その 2 - pathlib モジュール	123
2.7.8.1	パスオブジェクトの生成	124
2.7.8.2	カレントディレクトリ、ホームディレクトリの取得	124
2.7.8.3	パスの存在の検査	124
2.7.8.4	ファイル、ディレクトリの検査	124
2.7.8.5	ディレクトリの要素を取得する	124
2.7.8.6	ディレクトリ名、ファイル名、拡張子の取り出し	124
2.7.8.7	パスの連結	125
2.7.8.8	ファイルシステム毎のパスの表現	125
2.7.8.9	URI への変換	125
2.7.8.10	ファイルのオープン	125
2.7.8.11	ファイル入出力	126
2.7.8.12	ディレクトリの作成	127
2.7.8.13	ファイル、ディレクトリの削除	127
2.7.8.14	Path オブジェクトの文字列への変換	127
2.7.9	コマンド引数の取得	128
2.7.10	入出力処理の際に注意すること	128
2.7.11	CSV ファイルの取り扱い：csv モジュール	128
2.7.11.1	CSV ファイルの出力	129
2.7.11.2	CSV ファイルの入力	132
2.8	関数の定義	137
2.8.1	引数について	137
2.8.1.1	引数に暗黙値（デフォルト値）を設定する方法	138
2.8.1.2	仮引数のシンボルを明に指定する関数呼び出し（キーワード引数）	138
2.8.1.3	引数の個数が不定の関数	139
2.8.1.4	実引数に '*' を記述する方法	139
2.8.1.5	キーワード引数を辞書として受け取る方法	139
2.8.1.6	実引数に '**' を記述する方法	140
2.8.1.7	引数に与えたオブジェクトに対する変更の影響	140
2.8.1.8	引数に関するその他の事柄	142
2.8.2	変数のスコープ（関数定義の内外での変数の扱いの違い）	146
2.8.3	関数の再帰的定義	147
2.8.3.1	再帰的呼び出しの回数の上限	147
2.8.3.2	再帰的呼び出しを応用する際の注意	148
2.8.4	内部関数	149
2.8.4.1	内部関数におけるローカル変数	150
2.8.4.2	内部関数はいつ作成されるのか	150
2.8.4.3	内部関数の応用：クロージャ、エンクロージャ	151
2.8.5	定義した関数の削除	153
2.9	オブジェクト指向プログラミング	154
2.9.1	クラスの定義	154
2.9.1.1	コンストラクタとファイナライザ	154
2.9.1.2	インスタンス変数	156

2.9.1.3	メソッドの定義	156
2.9.1.4	クラス変数	157
2.9.2	クラス、インスタンス間での属性の関係	159
2.9.3	setattr, getattr による属性へのアクセス	160
2.9.4	組み込みの演算子や関数との連携	161
2.9.4.1	各種クラスのコンストラクタに対するフック	162
2.9.4.2	算術演算子, 比較演算子, 各種関数に対するフック	163
2.9.4.3	累算代入演算子に対するフック	165
2.9.5	インスタンス変数へのアクセスのカスタマイズ	165
2.9.5.1	存在しないインスタンス変数が参照された際のフック: <code>__getattr__</code>	165
2.9.5.2	インスタンス変数の参照に対するフック: <code>__getattribute__</code>	166
2.9.5.3	インスタンス変数への値の代入に対するフック: <code>__setattr__</code>	167
2.9.6	コンテナのクラスを実装する方法	167
2.9.6.1	マルチスライス (多重スライス)	168
2.9.7	イテレータのクラスを実装する方法	169
2.9.8	カプセル化	170
2.9.8.1	変数の隠蔽	170
2.9.8.2	アクセサ (ゲッター, セッター)	171
2.9.9	メソッドのオーバーライドと <code>super()</code>	173
2.9.9.1	<code>super</code> オブジェクト	174
2.9.10	インスタンス生成に関する制御: <code>__new__</code>	176
2.9.10.1	新規インスタンス生成の仕組み	176
2.9.10.2	シングルトン	177
2.9.11	オブジェクト指向プログラミングに関するその他の事柄	178
2.9.11.1	クラス変数, インスタンス変数の調査	178
2.9.11.2	クラスの継承関係の調査	178
2.9.11.3	インスタンスのクラスの調査	179
2.9.11.4	属性の調査 (プロパティの調査)	179
2.9.11.5	多重継承におけるメソッドの優先順位	180
2.9.11.6	静的メソッド (スタティックメソッド)	181
2.9.11.7	クラスの定義の削除	181
2.10	データ構造に則したプログラミング	183
2.10.1	map 関数	183
2.10.1.1	複数の引数を取る関数の map	185
2.10.1.2	map 関数に zip オブジェクトを与える方法	185
2.10.2	lambda と関数定義	186
2.10.3	filter	187
2.10.4	3 項演算子としての <code>if~else...</code>	188
2.10.5	all, any による一括判定	188
2.10.5.1	使用上の注意: ネストされたデータ構造	190
2.10.6	高階関数モジュール: <code>functools</code>	190
2.10.6.1	reduce	190
2.11	代入式	191
2.12	構造的パターンマッチング	192
2.12.1	構造的なパターン	193
2.12.2	条件付きのパターンマッチング	194
2.12.3	クラスのインスタンスに対するパターンマッチング	194

3 Kivy による GUI アプリケーションの構築	196
3.1 Kivy の基本	196
3.1.1 アプリケーションプログラムの実装	196
3.1.2 GUI 構築の考え方	196
3.1.2.1 Widget (ウィジェット)	197
3.1.2.2 Layout (レイアウト)	197
3.1.2.3 Screen (スクリーン)	199
3.1.3 ウィンドウの扱い	200
3.1.4 マルチタッチの無効化	200
3.2 基本的な GUI アプリケーション構築の方法	201
3.2.1 イベント処理 (導入編)	201
3.2.1.1 イベントハンドリング	201
3.2.2 アプリケーション構築の例	202
3.2.3 イベント処理 (コールバックの登録による方法)	205
3.2.4 ウィジェットの登録と削除	206
3.2.5 アプリケーションの開始と終了のハンドリング	207
3.3 各種ウィジェットの使い方	207
3.3.1 ラベル: Label	208
3.3.1.1 リソースへのフォントの登録	209
3.3.2 ボタン: Button	210
3.3.3 テキスト入力: TextInput	210
3.3.4 チェックボックス: CheckBox	211
3.3.5 進捗バー: ProgressBar	211
3.3.6 スライダ: Slider	211
3.3.7 スイッチ: Switch	211
3.3.8 トグルボタン: ToggleButton	212
3.3.9 画像: Image	212
3.3.9.1 サンプルプログラム	212
3.4 Canvas グラフィックス	213
3.4.1 Graphics クラス	214
3.4.1.1 Color	214
3.4.1.2 Line	214
3.4.1.3 Rectangle	214
3.4.1.4 Ellipse	215
3.4.2 サンプルプログラム	215
3.4.2.1 正弦関数のプロット	215
3.4.2.2 各種図形, 画像の表示	215
3.4.3 フレームバッファへの描画	217
3.4.3.1 ピクセル値の取り出し	218
3.4.3.2 イベントから得られる座標位置	218
3.5 スクロールビュー (ScrollView)	219
3.5.1 ウィジェットのサイズ設定	220
3.5.2 マウスのドラッグによるスクロール	220
3.6 ウィンドウサイズを固定 (リサイズを禁止) する設定	220
3.7 Kivy 言語による UI の構築	220
3.7.1 Kivy 言語の基礎	221
3.7.1.1 サンプルプログラムを用いた説明	221
3.7.1.2 Python プログラムと Kv ファイルの対応	223

3.8	時間によるイベント	224
3.8.1	時間イベントのスケジュール	224
3.9	GUI 構築の形式	224
3.9.1	スクリーンの扱い: Screen と ScreenManager	224
3.9.1.1	ScreenManager	224
3.9.1.2	Screen	225
3.9.2	アクションバー: ActionBar	226
3.9.3	タブパネル: TabbedPanel	228
3.9.4	スワイプ: Carousel	229
4	実用的なアプリケーション開発に必要な事柄	231
4.1	日付と時間に関する処理	231
4.1.1	日付と時刻の取り扱い: datetime モジュール	231
4.1.1.1	datetime オブジェクトの分解と合成	231
4.1.1.2	文字列を datetime オブジェクトに変換する方法	232
4.1.1.3	datetime オブジェクトの差: timedelta	232
4.1.1.4	日付, 時刻の書式整形	233
4.1.1.5	datetime のプロパティ	233
4.1.1.6	タイムゾーンについて	234
4.1.2	time モジュール	237
4.1.2.1	基本的な機能	237
4.1.2.2	日付, 時刻を表す文字列表現	238
4.1.2.3	応用例: datetime オブジェクトとの間の変換	239
4.1.2.4	時間の計測	240
4.1.2.5	プログラムの実行待ち	242
4.1.3	timeit モジュール	242
4.1.3.1	計測方法の指定	243
4.2	ロケール (locale)	244
4.2.1	ロケール設定の確認	244
4.2.2	使用できるロケールの調査	245
4.2.3	その他	246
4.2.3.1	現在のエンコーディングの調査	246
4.2.3.2	通貨の書式整形	246
4.3	文字列検索と正規表現	247
4.3.1	パターンの検索	247
4.3.1.1	正規表現を用いた検索	249
4.3.1.2	検索パターンの和結合	251
4.3.1.3	正規表現を用いたパターンマッチ	251
4.3.1.4	行頭や行末でのパターンマッチ	252
4.3.2	置換処理: re.sub	253
4.3.2.1	複数行に渡る置換処理	253
4.3.2.2	パターンマッチのグループを参照した置換処理	254
4.3.3	文字列の分解への応用: re.split	254
4.4	マルチスレッドとマルチプロセス	255
4.4.1	マルチスレッド	255
4.4.1.1	スレッドの実行状態の確認	256
4.4.1.2	実行中のスレッドのリストを取得する方法	257
4.4.1.3	スレッド間の排他制御	257

4.4.2	マルチプロセス	260
4.4.2.1	プロセスごとに実行されるモジュール	261
4.4.2.2	プロセスの実行状態の確認	261
4.4.2.3	実行中プロセスのリストを取得する方法	262
4.4.2.4	プロセスの強制終了	262
4.4.2.5	共有メモリと排他制御	263
4.4.3	concurrent.futures	270
4.4.3.1	ProcessPoolExecutor	270
4.4.3.2	プロセスの実行状態と戻り値	270
4.4.3.3	同時に実行されるプロセスの個数について	272
4.4.3.4	プロセス終了の同期	272
4.4.4	マルチスレッドとマルチプロセスの実行時間の比較	273
4.5	非同期処理：asyncio	275
4.5.1	コルーチン	275
4.5.2	コルーチンとタスクの違い	276
4.5.2.1	複数のタスクの並行実行	276
4.5.3	await によるタスクの一時停止と切替え	276
4.5.4	タスク登録の簡便な方法	277
4.5.5	イベントループと run 関数	278
4.5.5.1	スレッド毎に実行されるイベントループ	278
4.5.5.2	実行中のイベントループを調べる方法	279
4.5.6	理解を深めるためのサンプル	279
4.5.6.1	非同期処理とイベントループの概観	280
4.5.7	スレッドを非同期処理で管理する方法	281
4.5.8	非同期のコンソール入力を実現するためのライブラリ：aioconsole	283
4.5.9	永続するイベントループ上でのタスク管理	284
4.5.9.1	永続するイベントループ	284
4.5.9.2	タスクの一覧取得とタスクのキャンセル	285
4.5.10	非同期のイテレーション	287
4.5.10.1	非同期のイテラブル	287
4.5.11	非同期のファイル入出力：aiofiles	288
4.5.11.1	サンプルプログラム	289
4.6	処理のスケジューリング	291
4.6.1	sched モジュール	291
4.6.1.1	基本的な使用方法	291
4.6.1.2	イベントを独立したスレッドで実行する方法	292
4.6.1.3	イベントの管理	294
4.6.2	schedule モジュール	294
4.6.2.1	一定の時間間隔で処理を起動する方法	295
4.6.2.2	指定した時刻に処理を起動する方法	296
4.7	ジェネレータ	297
4.7.1	ジェネレータ関数	297
4.7.2	ジェネレータ式	298
4.7.3	ジェネレータの入れ子	298
4.8	モジュール、パッケージの作成による分割プログラミング	299
4.8.1	モジュール	299
4.8.1.1	単体のソースファイルとしてのモジュール	299
4.8.2	パッケージ（ディレクトリとして構成するライブラリ）	300

4.8.2.1	モジュールの実行	301
4.8.3	モジュールが配置されているディレクトリの調査	301
4.8.4	<code>__init__.py</code> について	302
4.9	ファイル内でのランダムアクセス	304
4.9.1	ファイルのアクセス位置の指定 (ファイルのシーク)	304
4.9.2	サンプルプログラム	304
4.10	オブジェクトの保存と読み込み: <code>pickle</code> モジュール	306
4.10.1	シリアライズと復元のカスタマイズ方法	308
4.10.2	シリアライズされたバイナリデータのプロトコルレベル	309
4.10.2.1	シリアライズされたバイナリデータのプロトコルレベルを調べる方法	310
4.10.2.2	シリアライズ時のプロトコルレベルの指定	310
4.10.3	使用上の注意事項	312
4.11	バイナリデータの作成と展開: <code>struct</code> モジュール	313
4.11.1	バイナリデータの作成	313
4.11.2	バイナリデータの展開	314
4.11.3	バイトオーダーについて	314
4.12	バイナリデータをテキストに変換する方法: <code>base64</code> モジュール	316
4.12.1	バイト列→Base64 データ	316
4.12.2	Base64 データ→バイト列	316
4.12.3	サンプルプログラム	316
4.13	編集可能なバイト列: <code>bytearray</code>	317
4.13.1	<code>bytearray</code> の作成方法	317
4.13.2	他の型への変換	318
4.13.3	ファイルへの出力	318
4.14	メモリ上でのファイル操作	319
4.14.1	仮想的なテキストファイル: <code>io.StringIO</code>	319
4.14.2	仮想的なバイナリファイル: <code>io.BytesIO</code>	319
4.14.3	<code>StringIO</code> , <code>BytesIO</code> を使用する際の注意事項	320
4.15	<code>exec</code> と <code>eval</code>	321
4.15.1	名前空間の指定	321
4.15.2	<code>eval</code> 関数	322
4.16	<code>collections</code> モジュール	323
4.16.1	キュー: <code>deque</code>	323
4.16.1.1	要素の追加と取り出し: <code>append</code> , <code>pop</code>	323
4.16.1.2	要素の順序の回転: <code>rotate</code>	323
4.16.2	要素の集計: <code>Counter</code>	324
4.16.2.1	出現頻度の順に集計結果を取り出す	324
4.16.3	<code>namedtuple</code>	325
4.17	<code>itertools</code> モジュール	326
4.17.1	イテラブルの連結: <code>chain</code>	326
4.17.2	無限のカウンタ: <code>count</code>	326
4.17.3	イテラブルの繰り返し: <code>cycle</code>	327
4.17.4	オブジェクトの繰り返し: <code>repeat</code>	327
4.17.5	連続要素のグループ化: <code>groupby</code>	327
4.17.6	直積集合: <code>product</code>	328
4.17.7	組合せ: <code>combinations</code>	329
4.17.8	順列: <code>permutations</code>	329
4.18	列挙型: <code>enum</code> モジュール	330

4.18.1 Enum 型	330
4.18.1.1 class 定義による Enum オブジェクトの作成	331
4.18.2 定数の取り扱いを実現する方法の例	331
4.18.3 IntEnum	332
4.18.4 auto 関数による Enum 要素への値の割り当て	333
4.18.4.1 非数値要素への auto 関数による値の割り当て (参考事項)	333
4.18.5 ビットフラグ: Flag	334
4.18.5.1 Flag の要素への auto 関数による値の割り当て	335
4.18.5.2 整数の性質を持つビットフラグ: IntFlag	335
4.19 例外 (エラー) の処理	337
4.19.1 エラーメッセージをデータとして取得する方法: traceback モジュール	337
4.19.2 例外を発生させる方法	337
4.19.3 例外オブジェクト	338
4.19.3.1 例外オブジェクトのクラス階層	339
4.20 使用されているシンボルの調査	339
4.21 with 構文	340
4.22 デコレータ	344
4.22.1 引数を取るデコレータ	345
4.23 データ構造の整形表示: pprint モジュール	346
4.24 文字列の整形処理 (分割, 折り返しなど): textwrap モジュール	347
4.24.1 長い文字列の分割	347
4.24.2 インデントの除去	348
4.24.3 インデントの挿入	348
4.25 処理環境に関する情報の取得	349
4.25.1 Python のバージョン情報の取得	349
4.25.2 platform モジュールの利用	349
4.25.3 実行中のプログラムの PID の取得	350
4.25.4 環境変数の参照	350
4.26 ファイル, ディレクトリに対する操作: shutil モジュール	352
4.26.1 ファイルの複製	352
4.26.2 ディレクトリ階層の複製	352
4.26.3 書庫ファイル (アーカイブ) の取り扱いと圧縮処理に関すること	352
4.26.3.1 書庫 (アーカイブ) の作成	353
4.26.3.2 書庫 (アーカイブ) の展開	353
4.27 ZIP 書庫の扱い: zipfile モジュール	353
4.27.1 ZIP 書庫ファイルを開く	353
4.27.2 書庫へのメンバの追加	354
4.27.3 書庫の内容の確認	354
4.27.4 書庫のメンバの読み込み	354
4.27.5 書庫の展開	355
4.27.5.1 パスワードで保護された ZIP 書庫へのアクセス	355
4.28 コマンド引数の扱い: argparse モジュール	356
4.28.1 コマンド引数の形式	356
4.28.2 使用方法	356
4.28.2.1 オプション引数の設定	356
4.28.2.2 位置引数の設定	357
4.28.2.3 コマンドラインの解析処理	357
4.28.3 サンプルプログラムに沿った説明	357

4.28.3.1	ヘルプ機能	358
4.28.3.2	コマンド引数の型の指定	358
4.28.4	サブコマンドの実現方法	359
4.29	スクリプトの終了（プログラムの終了）	360
4.29.1	sys.exit 関数	360
4.29.2	os._exit 関数	361
4.29.3	スクリプト終了時に実行する処理：atexit モジュール	361
4.30	Python の型システム	362
4.30.1	type 型オブジェクト	362
4.30.2	型の階層（クラス階層）	362
4.30.2.1	スーパークラス、サブクラスを調べる方法	363
4.30.3	数のクラス階層：numbers モジュール	364
5	TCP/IP による通信	367
5.1	socket モジュール	367
5.1.1	ソケットの用意	367
5.1.2	サーバ側プログラムの処理	367
5.1.3	クライアント側プログラムの処理	368
5.1.4	送信と受信	368
5.1.5	サンプルプログラム：最も基本的な通信	368
5.1.6	実用的な通信機能を実現する方法	370
5.1.6.1	socket ライブラリと asyncio ライブラリの併用	370
5.1.6.2	asyncio ライブラリのみで通信を実現する方法	372
5.1.6.3	通信の処理において注意すべき例外	375
5.2	WWW コンテンツ解析	376
5.2.1	requests ライブラリ	376
5.2.1.1	リクエストの送信に関するメソッド	376
5.2.1.2	取得したコンテンツに関するメソッド	376
5.2.1.3	Session オブジェクトに基づくアクセス	377
5.2.2	Beautiful Soup ライブラリ	378
5.2.2.1	BS における HTML コンテンツの扱い	378
6	外部プログラムとの連携	380
6.1	外部プログラムを起動する方法	380
6.1.1	標準入出力の接続	380
6.1.1.1	外部プログラムの標準入力のカローズ	383
6.1.2	非同期の入出力	383
6.1.3	外部プロセスとの同期（終了の待機）	385
6.1.4	外部プログラムを起動する更に簡単な方法	385
7	サウンドの入出力	387
7.1	基礎知識	387
7.2	WAV 形式ファイルの入出力：wave モジュール	387
7.2.1	WAV 形式ファイルのオープンとクローズ	387
7.2.1.1	WAV 形式データの各種属性について	388
7.2.2	WAV 形式ファイルからの読み込み	388
7.2.3	サンプルプログラム	388
7.2.4	量子化ビット数とサンプリング値の関係	389
7.2.5	読み込んだフレームデータの扱い	389

7.2.6	WAV 形式データを出力する例 (1)：リストから WAV ファイルへ	391
7.2.7	WAV 形式データを出力する例 (2)：NumPy の配列から WAV ファイルへ	392
7.2.8	サウンドのデータサイズに関する注意点	392
7.3	サウンドの入力と再生：PyAudio ライブラリ	394
7.3.1	ストリームを介したサウンド入出力	394
7.3.2	WAV 形式サウンドファイルの再生	395
7.3.2.1	サウンド再生の終了の検出	397
7.3.3	音声入力デバイスからの入力	397
A	Python に関する情報	400
A.1	Python のインターネットサイト	400
A.2	Python のインストール作業の例	400
A.2.1	PSF 版インストールパッケージによる方法	400
A.2.2	Anaconda による方法	401
A.3	Python 起動のしくみ	402
A.3.1	PSF 版 Python の起動	402
A.3.2	Anaconda Navigator の起動	402
A.3.3	Anaconda Prompt の起動	402
A.3.3.1	conda コマンドによる Python 環境の管理	404
A.4	PIP によるライブラリ管理	404
A.4.1	PIP コマンドが実行できない場合の解決策	405
B	Kivy に関する情報	406
B.1	Kivy 利用時のトラブルを回避するための情報	406
B.1.1	Kivy が使用する描画 API の設定	406
B.1.2	SDL について	406
B.2	GUI デザインツール	406
C	Tkinter：基本的な GUI ツールキット	407
C.1	基本的な扱い方	407
C.1.1	使用例	408
C.1.1.1	ウィンドウサイズ変更の可否の設定	409
C.1.2	ウィジェットの配置	409
C.2	各種のウィジェット	411
C.2.1	チェックボタンとラジオボタン	411
C.2.1.1	Variable クラス	411
C.2.2	エントリ（テキストボックス）とコンボボックス	413
C.2.2.1	プログラムの終了	414
C.2.3	リストボックス	414
C.2.4	テキスト（文字編集領域）とスクロールバー	416
C.2.5	スケール（スライダ）とプログレスバー	417
C.2.5.1	Variable クラスのコールバック関数設定	418
C.3	メニューの構築	419
C.4	Canvas の描画	420
C.4.1	描画メソッド（一部）	420
C.4.1.1	使用できるフォントを調べる方法	421
C.4.2	図形の管理	423
C.5	イベントハンドリング	425
C.5.1	時間を指定した関数の実行	427

C.6	複数のウィンドウの表示	428
C.7	ディスプレイやウィンドウに関する情報の取得	429
C.8	メッセージボックス (messagebox)	430
C.8.1	アプリケーション終了のハンドリング	431
C.9	ウィジェットの親、子を調べる方法	432
D	ライブラリの取り扱いについて	433
D.1	ライブラリの読み込みに関すること	433
D.1.1	ライブラリ読み込みにおける別名の付与	433
D.1.2	接頭辞を省略するためのライブラリの読み込み	433
D.1.2.1	接頭辞を省略する際の注意 (名前の衝突)	434
D.1.3	ライブラリのパス: sys.path	434
D.1.4	既に読み込まれているライブラリの調査: sys.modules	434
D.1.5	同一ライブラリの複数回の読み込みに関すること	435
D.1.6	The Zen of Python	436
D.2	ライブラリ情報の取得	436
D.2.1	pkg_resources による方法	436
D.2.2	pkgutil, importlib による方法	437
D.3	各種ライブラリの紹介	439
E	対話モードを使いやすくするための工夫	440
E.1	警告メッセージの抑止と表示	440
E.2	メモリの使用状態の管理	441
E.2.1	オブジェクトのサイズの調査	441
F	文書化文字列と関数アノテーション	443
F.1	関数アノテーション (Function Annotations)	444
G	型ヒント	445
G.1	型ヒントの存在意義	445
G.2	mypy によるプログラムの静的診断	446
G.3	変数に対する型ヒント	446
G.4	型ヒントのためのライブラリ: typing モジュール	447
G.4.1	様々な型ヒント	448
H	スロットベースのクラス	449
H.1	スロットベースのクラスが役立つケース	450
H.2	スロットベースのクラスを使用する上での注意点	450
I	サンプルプログラム	453
I.1	リスト／セット／辞書のアクセス速度の比較	453
I.1.1	スライスに整数のインデックスを与える形のアクセス	453
I.1.2	メンバシップ検査に要する時間	454
I.2	ライブラリ使用の有無における計算速度の比較	457
I.3	os.walk 関数の応用例	460
I.4	pathlib の応用例	461
I.5	浮動小数点数と 2 進数の間の変換	462
I.5.1	mpmath を用いた例	463
I.6	全ての Unicode 文字の列挙	464

J	その他	465
J.1	演算子の優先順位	465
J.2	アスキーコード表	466

1 はじめに

Python はオランダのプログラマであるガイド・ヴァンロッサム（Guido van Rossum）によって 1991 年に開発されたプログラミング言語であり、言語処理系は基本的にインタプリタである。Python は多目的の高水準言語であり、言語そのものの習得とアプリケーション開発に要する労力が比較的少ないとされる。しかし、実用的なアプリケーションを開発するために必要とされる多くの機能が提供されており、この言語の有用性の評価が高まっている。

Python の言語処理系（インタプリタ）は多くのオペレーティングシステム（OS）に向けて用意されている。Python で記述されたプログラムはインタプリタ上で実行されるために、C や Java と比べて実行の速度は遅いが、C 言語などの言語で開発されたプログラムとの連携が容易である。このため、Python に組み込んで使用するための各種の高速なプログラムがライブラリ（モジュール、パッケージ）の形で多数提供されており、それらを利用することができる。Python 用のライブラリとして利用できるものは幅広く、言語そのものの習得や運用の簡便性と相俟って、情報工学や情報科学とは縁の遠い分野の利用者に対してもアプリケーション開発の敷居を下げている。

本書で前提とする Python の版は 3（3.6 以降）である。

1.1 Python でできること

Python は汎用のプログラミング言語である。また、世界中の開発コミュニティから多数のライブラリが提供されている。このため、Python が利用できる分野は広く、特に科学、工学系分野のためのライブラリが豊富である。それらライブラリを利用することで、情報処理技術を専門としない領域の利用者も手軽に独自の情報処理を実現することができる。特に次に挙げるような処理を実現する上で有用なライブラリが揃っている。

- 機械学習
- ニューラルネットワーク
- データサイエンス
- GUI / Web アプリケーション開発

1.2 本書の内容

本書は Python でアプリケーションプログラムを開発するために必要な最小限の事柄について述べる。本書の読者としては、Python 以外の言語でプログラミングやスクリプティングの基本を学んだ人が望ましい。本書の内容を概略として列挙すると次のようなものとなる。

- Python 処理系の導入の方法
インタプリタとライブラリのインストール方法など
- Python の言語としての基本事項
文法など
- 実用的なアプリケーションを作成するために必要なこと
通信やマルチスレッド、マルチプロセスプログラミングに関すること
特に重要なライブラリに関すること
- GUI アプリケーションを作成するために必要なこと
GUI ライブラリの基本的な扱い方など
- サウンドの基本的な扱い
WAV ファイルの入出力や音声のサンプリングと再生

1.3 本書の読み方

Python を初めて学ぶ方には、言語としての基礎的な内容について解説している章

「1 はじめに」（この章）

「2 Python の基礎」

を読むことをお勧めする。その後、実用的なプログラミングにおいて重要となる事柄に関して解説した章

「4 実用的なアプリケーション開発に必要な事柄」

を読むことをお勧めする。その他の章については、学びの初期においては必ずしも必要とはならないこともあり、読者の事情に合わせて利用されたい。

1.4 処理系の導入（インストール）と起動の方法

Python の処理系（インタプリタ）は **Python ソフトウェア財団**（以後 PSF と略す）が開発、維持しており、当該財団の公式インターネットサイト <https://www.python.org/> から入手することができる。Microsoft 社の Windows や Apple 社の Mac ¹ には専用のインストールパッケージが用意されており、それらを先のサイトからダウンロードしてインストールすることで Python 処理系が使用可能となる。Linux をはじめとする UNIX 系 OS においては、OS のパッケージ管理機能を介して Python をインストールできる場合が多いが、先のサイトから Python のソースコードを入手して処理系をビルドすることもできる。

1.4.1 Python 処理系のディストリビューション（配布形態）

Python 処理系は C 言語により記述されており、C 言語処理系によってビルドすることができるが、コンパイル済みのインストールパッケージ（前述）を利用するのが便利である。インストールパッケージとしては PSF が配布するもの以外に Anaconda, Inc.² が配布する Anaconda が有名である。

PSF のインストールパッケージや Anaconda で Python 処理系を導入すると、各種のライブラリを管理するためのツールも同時に導入されるので、Python を中心としたソフトウェア開発環境を整えるための利便性が高い。ただし注意すべき点として、PSF の管理ツールと Anaconda ではライブラリの管理方法や、Python 自体のバージョン管理の方法が異なるため、Python の処理環境を導入する際は、ディストリビューション（PSF か Anaconda か）を統一すべきである。

※ 巻末付録「A.2 Python のインストール作業の例」（p.400）でインストール方法を概略的に紹介している。

参考. Windows 環境で UNIX 系ツールを利用するための MSYS/MinGW があるが、MSYS/MinGW 環境下でも Python 処理系を利用することができる。ただし、本書では MSYS/MinGW については触れない。

本書では PSF のインストールパッケージによって Python を導入した処理環境を前提とするが、言語としての Python の文法はディストリビューションに依らず共通であり、各種のライブラリも導入済みの状態であれば、使用方法は原則として同じである。

1.4.2 Python 処理系の起動

基本的に Python は OS のコマンドを投入することで起動する。Windows ではコマンドプロンプトウィンドウから `py` コマンド（PSF 版 Python）を、Linux や Mac ではターミナルウィンドウから `python` コマンドを投入³ することで次の例のように Python インタプリタが起動する。

例. Windows のコマンドプロンプトから Python を起動する例

```
C:\¥Users¥katsu> py Enter      ← py コマンドの投入
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>      ← Python プログラムの入力待ち
```

注 1) 本書で例示する操作を Anaconda 環境下で実行するには、巻末付録「A.3.3 Anaconda Prompt の起動」（p.402）に示す方法で Python 処理系を起動する。

注 2) 本書では JupyterLab や IPython, Google Colaboratory ⁴ といった対話環境については言及しない。JupyterLab の基本的な使用方法に関しては別冊の「Python3 ライブラリブック」で解説しているのでそちらを参照のこと。

¹Apple 社の Macintosh 用の OS の呼称は「Mac OS X」, 「OS X」, 「macOS」と変化している。本書ではこの内「OS X」, 「macOS」を暗に前提としている。

²<https://www.anaconda.com/>

³Apple 社の Mac では Python 2.7 が基本的にインストールされている場合があり、`python` コマンドを投入するとこの版 (2.7) が起動することがある。本書で前提とする Python3 をインストールして起動する場合は `python3` コマンドを投入すると良い。ただし、詳細に関しては利用する環境の導入状態を調べる。Anaconda 環境では Anaconda Prompt 下で `python` コマンドを発行する。

⁴Google 社が提供するオンラインの Python 実行環境。

1.4.3 対話モード：REPL

先の例の最後の行で「>>>」と表示されているが、これは Python のプロンプトであり、これに続いて Python の文や式を入力することができる。この状態を Python の対話モードと呼ぶ。対話モードを REPL (Read-Eval-Print Loop) と呼ぶこともある。

1.4.3.1 Python 言語処理系の終了

Python 言語処理系を終了するには `exit()` と入力する。

例. Python の終了

```
>>> exit() Enter    ← Python 終了
C:\Users\katsu>    ← Windows のコマンドプロンプトに戻る
```

この他にも、`quit()` の実行、あるいは Ctrl+Z などの終了方法がある。また Python 3.13 の版からは `exit`, `quit` に括弧 `()` を付けずに入力して対話モード (REPL) を終了⁵ することができる。

対話モードでは、入力された Python の文や式が逐一実行されて結果が表示される。そして再度プロンプト「>>>」が表示されて、次の入力を受け付ける。また、キーボードのカーソルキー ↑, ↓, ←, → で入力途中の行の編集ができるだけでなく、入力の履歴を呼び出すこともできる。

1.4.3.2 ヒストリ

Python 処理系の対話モードでは、入力した文や式が記録されており、再度呼び出して使用することができる。例えば次のように `print` 関数を入力する場合を考える。

例. `print` 関数を 3 回実行する

```
>>> print('1 番目の入力') Enter
1 番目の入力
>>> print('2 番目の入力') Enter
2 番目の入力
>>> print('3 番目の入力') Enter
3 番目の入力
>>>                                ← 次の入力を促すプロンプト
```

この例では 3 つの `print` 関数を入力しているが、それらは履歴に記録されており、次の例のようにキーボードのカーソルキー ↑ を押すことで再度呼び出すことができる。

例. 実行済みの行を再度呼び出す (先の例の続き)

```
>>> ↑                                プロンプトに対して上矢印のカーソルキーを押すと
↓ ↓ ↓ ↓
>>> print('3 番目の入力')    直前の入力が再度表示される
```

このように ↑, ↓ で以前に入力されたものを呼び出すことができる。

テキストファイルに書き並べた Python の文や式をスクリプトとしてまとめて実行することもできるが、この方法については「2.1 スクリプトの実行」(p.5) で説明する。

異なる複数のバージョンの Python 処理系を導入し、それらを選択して起動することもできる。これに関しては「A.3 Python 起動のしくみ」(p.402) で解説する。

1.4.3.3 直前の値

値を返す式を対話モードで評価 (実行) した場合、その値はアンダースコア「_」に保持される。(次の例参照)

⁵スクリプトの実行 (p.5「2.1 スクリプトの実行」参照) においては `exit()`, `quit()` と括弧を付けねばならない。

例. 評価結果の値の保持

```
>>> 2 + 3 Enter    ←足し算の式を評価（足し算の実行）
5          ←評価結果（計算結果）
>>> _ Enter    ←上の値を保持する「_」を確認
5          ←値が保持されている
```

この場合のアンダースコア「_」は**変数**⁶であり、別の式の中で参照することができる。

例. 「_」を別の式の中で参照する（先の例の続き）

```
>>> _ * 2 Enter    ←先の評価結果の値を2倍する式
10        ←評価結果（計算結果）
>>> _ Enter    ←上の値を保持する「_」を確認
10        ←値が更新されている
```

1.4.3.4 REPL の機能に関すること

Python 3.13 から REPL の機能が大幅に改善されたが、進んだ REPL の機能を抑止して 3.12 以前の状態と同様の形で起動するには、OS の環境変数を `PYTHON_BASIC_REPL=1` と設定する。

REPL 機能の抑止：

Unix 系 (bash) の場合： `export PYTHON_BASIC_REPL=1`
Windows の cmd の場合： `set PYTHON_BASIC_REPL=1`

1.5 本書で取り扱う GUI ライブラリ

グラフィカルユーザインターフェース (GUI) を備えたアプリケーションプログラムを構築するには GUI を実現するためのプログラムライブラリを入手して用いる必要がある。GUI のためのライブラリには様々なものがあり、Python 処理系に標準的に提供されている Tkinter をはじめ、GNOME Foundation が開発した GTK+, Qt Development Frameworks が開発した Qt, Julian Smart 氏 (英) が開発した wxWidgets, Kivy Organization が開発した Kivy など多くのものが存在している。具体的には、それら GUI のライブラリを Python から使用するために必要となるライブラリを導入し、Python のプログラムから GUI ライブラリを呼び出すための API を介して GUI を構築する。

本書では、比較的新しく開発され、携帯情報端末 (スマートフォンやタブレットなど) でも動作する GUI アプリケーションを構築することができる Kivy を主として⁷ 取り上げる。

1.6 表記に関する注意事項

本書内では多数のサンプルプログラムや実行例を示すが、実行する計算機環境によって日本円記号「¥」がバックスラッシュ「\」として表示されることがある。あるいは、バックスラッシュの入力によって日本円記号が表示される場合がある。このことを留意して、適宜読み替えていただきたい。

1.7 Python に関する詳しい情報

本書で説明していない情報に関しては、参考となる情報源 (インターネットサイトなど) を巻末の付録に挙げるので、そちらを参照のこと。

⁶「変数」については後の「2.4 変数とデータの型」(p.8) で解説する。

⁷Tkinter に関しても、付録「C Tkinter：基本的な GUI ツールキット」(p.407) で簡単に紹介する。

2 Pythonの基礎

他のプログラミング言語の場合と同様に、Python でも**変数**、**制御構造**、**入出力**の扱いを基本とする。Python の文や式はインタプリタのプロンプトに直接与えること（対話モードでの実行）もできるが、文や式をテキストファイルに書き並べて（スクリプトとして作成して）まとめて実行することもできる。

2.1 スクリプトの実行

例えば次のようなプログラムが test01.py というファイル名のテキストデータとして作成されていたとする。

プログラム：test01.py

```
1 # coding: utf-8
2 print( '私の名前はPythonです。' )
```

これを実行するには OS のターミナルウィンドウから、

python test01.py (Mac, Linux, Anaconda の場合⁸)

と入力するか、あるいは、

py test01.py (Windows 用 PSF 版の場合)

と入力する。するとターミナルウィンドウに「私の名前は Python です。」と表示される。

このプログラム例 test01.py の中に print で始まる行がある。これは **print 関数** というもので端末画面（標準出力）への出力の際に使用する関数であり⁹、今後頻繁に使用する。

《 print 関数 》

書き方： print(出力内容)

「出力内容」はコンマ「,」で区切って書き並べることができ、それら内容はスペースで区切った形で出力される。この際の区切りを「sep='区切り'」で明に指定することができる。

出力処理の後は改行されるが「end="」（単引用符 2 つ）を与えると行末の改行出力を抑止できる。これは出力の終端処理を指定するものであり、終端に出力するものを「end='終端に出力する内容'」として明に与えることもできる。

‘end=’, ‘sep=’ は**キーワード引数**と呼ばれるものであるが、これに関しては「2.8 関数の定義」(p.137) で説明する。

注意 print は関数なので¹⁰ 出力内容を必ず括弧「()」で括る（引数として与える）必要がある。

print 関数の区切り出力に関するサンプルプログラムを printTest02.py に示す。

プログラム：printTest02.py

```
1 # coding: utf-8
2 print('print関数に複数の引数を与える例を示します。')
3 print('区切り文字','を','指定しない','例です。')
4 print('区切り文字を','コロンに','した','例です。', sep=':')
5 print('区切り文字','なしに','した','例です。', sep='')
```

このプログラムを実行すると次のように表示される。

```
print 関数に複数の引数を与える例を示します。
区切り文字 を 指定しない 例です。
区切り文字を:コロンに:した:例です。
区切り文字なしにした例です。
```

print 関数の行末処理に関するサンプルプログラムを printTest01.py に示す。

⁸python3 としなければならない場合もあるので注意されたい。

⁹print 関数はファイルに対する出力の機能も持つ。これに関しては後の「2.7.4.1 print 関数によるファイルへの出力」(p.116) で解説する。

¹⁰Python2 までは print は文であった事情から括弧は必要なかったが、Python3 からは仕様が変わり、print は関数となった。このため括弧が必須となる。

プログラム：printTest01.py

```
1 # coding: utf-8
2 print( 'これは' )
3 print( 'print関数が' )
4 print( '改行する様子の' )
5 print( 'テストです. ', end='\n\n' )
6 print( 'しかし, ', end='' )
7 print( 'キーワード引数 end=\'\' を与えると', end='' )
8 print( 'print関数は', end='' )
9 print( '改行しません. ' )
```

このプログラムを実行すると次のように表示される。

```
これは
print 関数が
改行する様子の
テストです.
```

しかし、キーワード引数 `end=''` を与えると `print` 関数は改行しません。

このプログラムの5行目に「`end='\n\n'`」という記述があるが、これは改行を意味するエスケープシーケンス¹¹であり、「`\n`」を2つ指定することで行末で2回改行する。

注) 表示する端末によってはバックスラッシュ「`\`」¹²が「`¥`」と表示されることがある。

2.1.1 プログラム中に記述するコメント

「`#`」で始まる部分は行末までがコメントと見なされ、Pythonの文とは解釈されずに無視される。ただし、先のプログラムの冒頭にある

```
# coding: utf-8
```

は、プログラム（スクリプトのファイル）を記述するための文字コード体系¹³（エンコーディング）を指定するものであり、コメント行が意味を成す特別な例である。エンコーディングとしては `utf-8` が一般的であるが、この他にも表1に示すようなエンコーディングを使用してプログラムを記述することができる。

表 1: 指定できるエンコーディング（一部）

エンコーディング	coding:の指定	エンコーディング	coding:の指定	エンコーディング	coding:の指定
UTF-8	utf-8	EUC	euc-jp	シフト JIS	shift-jis, cp932*
UTF-8(BOM 付き)	utf-8-sig	JIS	iso2022-jp		

* マイクロソフト標準キャラクタセットに基づく指定

`coding:`に指定するものは大文字／小文字の区別はなく、ハイフン「`-`」とアンダースコア「`_`」のどちらを使用してもよい¹⁴。また、エンコーディング指定のコメント行を記述する際、

```
# -*- coding: utf-8 -*-
```

と記述することがあるが、この「 `-*-`」は Emacs¹⁵ で用いられる表現であり、省略しても問題はない。

参考)

Python 言語処理系では、スクリプトファイルのデフォルトのエンコーディングは `utf-8` であり、`utf-8` で記述されたプログラムの冒頭では「`# coding:エンコーディング`」の記述を省略しても良い。

■ 文字列の中に記述する「`#`」

文字列の中に記述した「`#`」はコメントとは見做されない。

¹¹出力の制御に関するもの。詳しくは「2.4.7.1 エスケープシーケンス」(p.38)で解説する。

¹²Apple社のMacintoshの機種によっては `Option` + `¥` と入力しなければならないことがある。

¹³ここで言う「文字コード体系」とは、対象の文字とそれを計算機内部で表現するためのバイト列との対応を規定するものである。「文字コード体系」に関しては後の「2.4.7.14 文字コード、文字の種別に関すること」(p.47)で更に詳しく解説する。

¹⁴若干の例外もある。詳しくは PSF の公式インターネットサイトの「標準エンコーディング」に関する codecs モジュールのドキュメントを参照のこと。

¹⁵Emacs Lisp による編集が行える高機能なテキストエディタである。

例. 文字列中の「#」

```
>>> s = 'abc#def'  ←「#」の記述があるが…  
>>> print(s)   
abc#def ←コメントではなく文字列として扱われる
```

2.1.2 プログラムのインデント

Python ではプログラムを記述する際のインデント（行頭の空白）に特別な意味があり、不適切なインデントをしてはならない。例えば次のような複数の行からなるプログラム `test02.py` は正常に動作するが、不適切なインデントを施したプログラム `test02-2.py` は実行時にエラーとなる。

正しいプログラム：test02.py

```
1 # coding: utf-8  
2 print( '1.私の名前はPythonです. ' )  
3 print( '2.私の名前はPythonです. ' )  
4 print( '3.私の名前はPythonです. ' )
```

間違ったプログラム：test02-2.py

```
1 # coding: utf-8  
2 print( '1.私の名前はPythonです. ' )  
3     print( '2.私の名前はPythonです. ' )  
4 print( '3.私の名前はPythonです. ' )
```

`test02-2.py` を実行すると次のようになる。

```
File "C:\Users\¥katsu¥Python¥test02-2.py", line 3  
print( '2.私の名前はPythonです. ' )  
IndentationError: unexpected indent
```

インデントが持つ意味については「2.6 制御構造」(p.89) のところで解説する。

2.2 文を分割して／連結して記述する方法

1つの文や式を複数の行に分割して記述するには行末に「¥」¹⁶ を記述する。

例. 1つの式 $1+2+3$ を複数の行に分割して記述する例（対話モードでの例）

```
>>> 1 ¥   
... + ¥   
... 2 ¥   
... + ¥   
... 3   
6 ←計算結果
```

ただし、この記述方法は可読性を低下させる原因にもなるので推奨されておらず、次の例のように複数の行を括弧（）で括る形式の方が良い。

例. 上の例と同等の記述

```
>>> (1   
... +   
... 2   
... +   
... 3)   
6 ←計算結果
```

複数の文や式を1つの行に書き並べるにはセミコロン「;」で区切る。

¹⁶これは先に説明したバックスラッシュと同じもので、Apple 社の Macintosh の機種によっては + ¥ と入力する。

例. 2つの文を1行に記述する例（対話モードでの例）

```
>>> print('2つの文で',end='') ; print('表示しました. ') Enter
2つの文で表示しました. ←実行結果
```

ただし、この記述方法もあまり推奨されていないことを留意されたい。

2.3 文の記述の省略

何も実行しない文として `pass` を記述することができる。

例. 何もしない文

```
>>> pass Enter ←何もしない文
>>> ←何もせずに次のプロンプトが表示される
```

対話モードで `pass` を入力する意味は無いが、実際のプログラム開発では、コードの記述を一時的に棚上げしておくことがしばしばある。そのような場合に `pass` を記述しておくのが一般的である。特にスイートの記述（p.89『2.6.1.2「スイート」の概念』で解説する）を一時棚上げにする際に用いることが多い。

2.4 変数とデータの型

変数はデータを保持するものであり、プログラミング言語の最も基本的な要素である。変数の記号に値を割り当てるにはイコール「=」を使用する。

例. 変数記号 `x` に値 10 を割り当てる

```
x = 10
```

変数に割り当てる（代入する）値には数値（整数、浮動小数点数）や文字列といった様々な**型**がある。C言語やJavaでは、使用する変数はその使用に先立って明に宣言する必要があり、変数を宣言する際にデータの型を指定しなければならない。宣言した型以外のデータをその変数に割り当てることはできない。この様子を指して「C言語やJavaは**静的な型付け**の言語処理系である」と表現する。これに対してPythonは**動的な型付け**の言語処理系であり、1つの変数に割り当てることのできる値の型に制約は無い。すなわち、ある変数に値を設定した後で、別の型の値でその変数の内容を上書きすることができる。またPythonでは、変数の使用に先立って変数の確保を明に宣言する必要は無い。例えば次のような例について考える。

例. 変数への値の代入

```
>>> x = 2 Enter ←変数 x に整数の 2 を代入
>>> y = 3 Enter ←変数 y に整数の 3 を代入
>>> z = x + y Enter ← x と y の値を加算したものを変数 z に代入
>>> print(z) Enter ←変数 z の内容を出力する処理
5 ←出力結果
```

これはPythonの処理系を起動して変数 `x`, `y` に整数の値を設定し、それらの加算結果を表示している例である。変数への値の設定（代入）にはイコール記号「=」を使う。引き続いて次のようにPythonの文を与える。

例. 既存の変数に値を上書きする（先の例の続き）

```
>>> x = "2" Enter ←変数 x に文字列の値"2"を代入
>>> y = "3" Enter ←変数 y に文字列の値"3"を代入
>>> z = x + y Enter ← x と y の値を連結したものを変数 z に代入
>>> print(z) Enter ←変数 z の内容を出力する処理
23 ←出力結果
```

この例は `x`, `y` に代入された文字列を、加算ではなく連結するものであり、同じ変数に異なる型の値が上書きされることがわかる。ただし、変数に代入される値の型の扱い（演算など）に関しては厳しい扱いが求められる。例えば数値と文字列を加算しようとする次のような結果（エラー）となる。

例. 数値と文字列を加算する試み

```
>>> 12 + "34" Enter      ←数値と文字列の加算の試み
Traceback (most recent call last):      ←異なる型同士の演算が許されない17
  File "<stdin>", line 1, in <module>    旨のエラーメッセージ
TypeError: unsupported operand type(s) for +: 'int' and 'str'      ←'TypeError'
```

ある変数に割当てられている値を別の変数に代入することができる。

例. 変数の値を別の変数に割当てて

```
>>> a = 3 Enter      ←変数 a に割当てられた値を
>>> b = a Enter      ←変数 b に割当てることができる
>>> print(b) Enter      ←変数 b の値を確認
3
```

「b = a」で変数 b に対して変数 a の値を代入している。これに対して「a = 3」の右辺にあるように直接的に「3」と記述したものをリテラルという。

例. リテラル

```
5          : 整数リテラルの 5
'apple'    : 文字列リテラルの 'apple'
```

2.4.1 日本語の変数名の使用

Python3 では日本語の変数名が使用できる。

例. 日本語の変数名

```
>>> あ = 'いうえお' Enter      ←変数名に「あ」を使用
>>> あ Enter      ←変数「あ」の値を確認
'いうえお'      ←値が保持されている
```

ただし日本語の変数名は、プログラムの可読性の低下を招くリスクがあるので、あまり推奨されない。

2.4.2 変数名に関する注意

変数名として使用できない、あるいは使用してはならない名前があることに注意すること。変数名はアルファベットもしくは Unicode の全角文字から始める。あるいはアンダースコア '_' も変数名に含めて（先頭に使用して）も良い。ただしハイフン '-' は減算の記号であるため変数名には使用できない。その他、特殊記号も変数名に使用できないものが多いので注意すること。

例. 変数名に使用できる／できない記号

```
>>> _a = 12 Enter      ←アンダースコアを使用した変数名が使用できる
>>> print(_a) Enter
12
>>> n! = 34 Enter      ←感嘆符 '!' を使用した変数名は
File "<stdin>", line 1      使用できない（エラーとなる）
  n! = 34
  ^
SyntaxError: invalid syntax
```

Python の予約語¹⁸ は変数名には使用できない。

例. 予約語を変数名に使用する試み

```
>>> class = 'A' Enter      ←予約語「class」を変数名に使用すると…
File "<stdin>", line 1      ←エラー（例外）が発生する
  class = 'A'
  ^
SyntaxError: invalid syntax      ←文法エラーとなっている
```

¹⁷Python 以外では（例えば JavaScript など）、異なる型の値同士の演算が許される言語も存在する。

¹⁸予約語：Python 言語の基本的な文を構成する要素。

これは、予約語「class」を変数名として使用する試みの例であるが、このような場合はエラー（例外）が発生する。また予約語以外のものであっても、システムが標準的に提供するオブジェクトの名前（関数名など）は変数として使用してはならない。

例. 「print」を変数名として用いる試み

```
>>> print( 'Python' )  Enter    ← print 関数の使用
Python                  ← 正常に動作している

>>> print = 'あいうえお'  Enter    ← 「print」という語を変数名として値を与え、
>>> print( 'Python' )  Enter    ← print を関数名として使用すると…
Traceback (most recent call last):    ← エラー（例外）が発生する
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

これは、既存の print 関数の名前を変数名として使用し、それに値を代入した例である。特に注意すべきこととして、この段階ではエラー（例外）が発生しないという点が重要である。この例では、以後「print」は関数として使用できなくなるが、変数名となった記号「print」を del 文によって解放すると再度「print」は関数名として使用可能となる。（次の例）

例. 変数名としての「print」を解放する（先の例の続き）

```
>>> del print  Enter    ← 変数名「print」の解放
>>> print( 'Python' )  Enter    ← print を関数名として使用する
Python                  ← 正常に動作している
```

print 以外の関数についても、誤って変数名として使用した際は同様の方法で回復を試みる必要があるが、回復できない場合は Python 言語処理系を再起動しなければならない。

2.4.3 イコール「=」の連鎖（Multiple Assignment）

イコール「=」の連鎖によって、1つの値を複数の変数に割り当てることができる。

例. 変数 a, b, c に値 12 を設定する

```
>>> a = b = c = 12  Enter
```

変数への値の割り当て方法はこの他にもある。後の「2.5.8 データ構造に沿った値の割り当て（分割代入）」(p.85)で更に高度な方法について説明する。

2.4.4 変数の解放（廃棄）

値を保持している変数を解放する（変数を廃棄する）には del 文を使用する。（次の例）

例. 変数への値の設定と廃棄

```
>>> a = 123  Enter    ← 変数への値の設定
>>> a  Enter    ← 値の確認
123          ← 結果表示
>>> del a  Enter    ← 変数の解放（廃棄）
>>> a  Enter    ← 値の確認
Traceback (most recent call last):    ← エラーメッセージ
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined    ← 変数に何も設定されていない
```

これは、値の設定された変数 a が del 文によって解放される例である。Python では、値が設定されていない（未使用の）変数を参照しようとする上のようなエラー（NameError）が発生する。

変数記号が値を持っているかどうか（シンボルが定義されているか未定義か）を調べる方法については「4.20 使用されているシンボルの調査」(p.339)を参照のこと。

2.4.5 変数が値を保持する仕組みの概観

言語処理系が扱う値（データ）はコンピュータの主記憶上に確保された**オブジェクト**であり、変数に値を割当てる（代入する）行為は、当該オブジェクトに変数記号を結びつける処理であるといえることができる。また、データを保持するオブジェクトは具体的な値の情報の他に、その値のデータ型（type）を始めとする各種の情報も保持する。

Python 言語処理系では、データを保持するオブジェクトのデータ型を `type` 関数で調べる（p.52「2.4.12 型の検査」で説明する）ことができる。また、メモリ上のオブジェクトの識別情報（識別値¹⁹）を `id` 関数で調べることができる。（次の例）

例. オブジェクトのデータ型と識別情報を調べる

```
>>> x = 3.14  Enter  ←変数への値の割当て
>>> type(x)   Enter  ←変数が持つオブジェクトの値の型を調べる
<class 'float'>      ←浮動小数点数（float）であることがわかる
>>> id(x)     Enter  ←変数が持つオブジェクトの識別情報を調べる
2125365768816        ←当該オブジェクトのための一意の値
```

この例ではオブジェクトの識別値は 2125365768816 となっているがこれは一例であり、実際には Python 処理系がオブジェクト作成時に決定する。

上の例の実行後の変数とオブジェクトの概観を図 1 に示す。

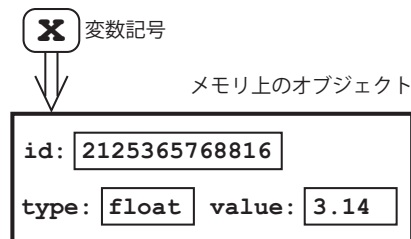


図 1: メモリ上のオブジェクトに変数記号が結び付けられている様子

同じ値のリテラルを別の変数に割当てると、基本的にそれは別のオブジェクトとして扱われる。（次の例）

例. 数値リテラル 3.14 を別の変数に割当てると（先の例の続き）

```
>>> y = 3.14  Enter  ←別の変数に先と同じリテラル値を割当てると
>>> type(y)   Enter
<class 'float'>
>>> id(y)     Enter  ←識別値は
2125365765232        別のものになっている
```

この処理の後の変数とオブジェクトの概観を図 2 に示す。

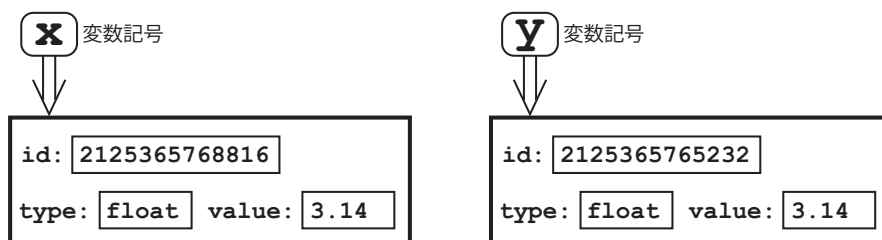


図 2: 変数 x, y はそれぞれ別のオブジェクトを指す

同じ 3.14 という値を持つ 2 つの変数 x, y はそれぞれ別のオブジェクトを指していることがわかる。しかし、既存の変数が持つ値を別の変数に割当てると、同一のオブジェクトを指すことに注意すべきである。（次の例）

¹⁹CPython 実装系では当該オブジェクトのメモリアドレスであり、システムが自動的に割り当てる。

例. 既存の変数の値を別の変数に割当てて (先の例の続き)

```
>>> z = x  ←既存の変数 x の値を別の変数 z に割当てて  
>>> type(z)   
<class 'float'>  
>>> id(z)  ←識別値は  
2125365768816 x と同じものになっている
```

この処理の概観を図 3 に示す。

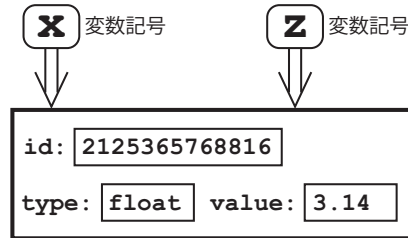


図 3: 変数 x, z は同一のオブジェクトを指す

1つのオブジェクトが2つの変数記号に結び付けられている様子がわかる。この様子を指して「オブジェクトの参照カウントが2である」という。参照カウントが0になったオブジェクトは基本的にはガベージコレクション (GC) の対象となり廃棄される。

2.4.5.1 識別値に関する注意事項

整数値のオブジェクトの扱いは浮動小数点数の場合と異なる。先の処理例において、変数 x, y に 3.14 の代わりに整数値を与えるケースを試みられたい。この他にも id 関数が返す識別値に関しては注意すべきことがいくつかある。

Python 処理系がプログラムを解釈する際、対話モードによる実行の場合とスクリプトの実行の場合で動作が異なることがある。次に示すサンプルプログラム `idTest01.py` について考える。

プログラム: `idTest01.py`

```
1 # coding: utf-8  
2 x = 3.14; y = 3.14 # 同じ浮動小数点数を別々の変数に代入  
3 print( 'x =',x, ', y= ',y )  
4 print( 'id(x) =',id(x) )  
5 print( 'id(y) =',id(y) )
```

このプログラムをスクリプトとして CPython 3.12.1 で実行すると次のような結果となる。

実行例. Windows 環境の CPython 3.12.1 による実行

```
C:\Users\katsu>py idTest01.py  ←コマンドプロンプトで実行  
x = 3.14 , y= 3.14  
id(x) = 2362509509744 ← x と y の識別値が  
id(y) = 2362509509744 同じになっている
```

対話モードで同様の処理を行った場合と結果が異なり、変数 x と y の識別値が同じになっていることがわかる。

上の例に示したように、実行時の状況や処理系の実装ごとに値やオブジェクトの識別値が異なることがあるので、id 関数の使用においては注意が必要である。

2.4.6 数値

Python では整数 (int 型)、浮動小数点数 (float 型)、複素数 (complex 型) を数として扱う。また、数に対する基本的な演算 (算術演算など) の表記を表 2 に示す。

参考) $a \div b$ における整数除算と剰余を同時に求める `divmod(a,b)` 関数もある。

注意) 整数除算, 剰余, `divmod` に関して詳しくは、後の「2.4.6.15 除算における商と剰余に関する事柄」(p.29) で解説する。

表 2: 数に対する基本的な演算 (算術演算など)

表記	意 味	表記	意 味
$a + b$	a と b の加算	$a - b$	減算
$a * b$	a と b の乗算	a / b	除算
$a // b$	整数除算	$a \% b$	剰余 (a を b で割った余り)
$a ** b$	冪乗 (a の b 乗 a^b)		

この他にも累算代入²⁰の演算子が使えらる。例えば,

a += b	は	a = a + b と同等,	a -= b	は	a = a - b と同等,
a *= b	は	a = a * b と同等,	a /= b	は	a = a / b と同等,
a %= b	は	a = a % b と同等,	a **= b	は	a = a ** b と同等
a //= b	は	a = a // b と同等			

といった演算である.

各種演算子の優先順位に関しては巻末付録「J.1 演算子の優先順位」(p.465)を参照のこと.

2.4.6.1 整数： int 型

整数としては長い桁の値が扱える. (次の例)

例. $2^{1,000}$ を計算する

```
>>> 2**1000 
```

107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837881569585
812759467291755314682518714528569231404359845775746985748039345677748242309854210746050623711418779
541821530464749835819412673987675591655439460770629145711964776865421676604298316526243868372056680
69376

補足) Python インタプリタのプロンプトに対して直接に計算式や値を入力して Enter を押すと、その式の計算結果（値）がそのまま出力される。

■ 整数値の表記上の注意

整数の値を記述する際、基本的には先頭に余分な 0 を記述してはならないが、00 の記述は問題ない。

例. 整数値の表記の注意

```
>>> 01  ←前に 0 を記述して 1 を入力する試み
File "<stdin>", line 1      ←エラーとなる
    01
    ^
SyntaxError: leading zeros in decimal integer literals are not permitted;
use an 0o prefix for octal integers

>>> 00  ←ただし、00 は
0      問題ない
```

2.4.6.2 巨大な整数値を扱う際の注意

巨大な（桁数の大きい）整数値を扱う際には注意しなければならないことがある。

例. $10^{4,299}$ の算出と出力

[illegible]

この例では 10^{4299} の値を計算しており、それが 4,299 個のゼロを含む 4,300 桁の値となることがわかる。次に 10^{4300} の値を計算する例を示す。

²⁰再帰代入と呼ぶこともある.

例. $10^{4,300}$ の算出と出力の試み

```
>>> x = 10**4300  [Enter]    ← 104300 の算出は可能、しかし…
>>> x  [Enter]    ← 値を出力して確認しようとする…
Traceback (most recent call last):                ← エラーとなる
  File "<stdin>", line 1, in <module>
ValueError: Exceeds the limit (4300) for integer string conversion;
use sys.set_int_max_str_digits() to increase the limit
```

巨大な整数値を算出することはできるが、それを出力しようとすると `ValueError` というエラーが発生することがこの例からわかる。またエラーメッセージの内容から、4,300 桁を超える整数値を出力する際にエラーが発生する²¹ ことがわかる。この上限値を超える桁数の値を出力するには `sys` モジュール²² の `set_int_max_str_digits` 関数を用いて出力桁数の上限値を変更すると良い。(次の例)

例. 出力桁数の上限値を大きくする（先の例の続き）

[illegible]

■ 整数値の桁数（10進数表現）を調べる方法

整数値の桁数（10進数表現）を調べるには対数関数を応用する方法がある。

例. 大きな整数値の桁数を調べる

```
>>> import math      Enter      ←必要なモジュールの読み込み
>>> x = 3**100      Enter      ←  $3^{100}$  の算出
>>> x      Enter      ←値の確認
515377520732011331036461129765621272702107522001      ←  $3^{100}$  の値
>>> math.floor(math.log10(abs(x)))+1      Enter      ←桁数を調べる
48      ← 48 桁であることがわかる
```

これは $x \neq 0$ の場合に $\lfloor \log_{10} |x| \rfloor + 1$ で整数部の桁数を算出する例で、使用した `math` モジュールに関しては後の「2.4.6.11 数学関数」(p.19)で解説する。

2.4.6.3 浮動小数点数： float 型

浮動小数点数は IEEE-754 で定められる倍精度 (double) で表現される。

例. $1.000000000001^{100,000,000,000}$ を求める

```
>>> 1.000000000001 ** 1000000000000 Enter
2.71828205335711 ←近似値が得られる（誤差を含んでいる）
```

float 型の値の記述には**指数表記**が使える。指数表記は非常に大きな数や、非常に小さな数を表現する際に用いられる表現であり、**仮数部** m と**指数部** n から成る $m \times 10^n$ という表記であるが、Python では、

m e n

と表記する.

例. 指数表記による小さな数, 大きな数の表記

```
>>> me = 9.1093837015e-31 Enter ←非常に小さな数
>>> NA = 6.02214076e23 Enter ←非常に大きな数
>>> print(' 電子の質量 (Kg):',me) Enter ←内容確認
電子の質量 (Kg): 9.1093837015e-31 ←結果表示
>>> print(' アボガドロ定数:',NA) Enter ←内容確認
アボガドロ定数: 6.02214076e+23 ←結果表示
```

²¹この制限は 2022 年 9 月に導入された. それ以前の版の Python ではこの制限はない.

²²システムパラメータの取得や設定などに関するモジュール

注) 値の大小に関係なく指数表記は使用できる.

例. 指数表記の例

```
>>> 0.31415e1  Enter    ← 0.31415 × 101
3.1415          ← 結果表示
```

■ float 型に関する各種の情報

float 型で扱える値には各種の制限があり, それらに関する情報は sys モジュール²³ を用いて参照することができる. float で扱える正の最大値は float_info.max, 正の最小値は float_info.min を参照すると得られる.

例. float 型で扱える値の最大値と最小値

```
>>> import sys  Enter    ← sys モジュールの読み込み
>>> sys.float_info.max  Enter    ← float 型で扱える正の最大値
1.7976931348623157e+308  ←これが正の最大値
>>> sys.float_info.min  Enter    ← float 型で扱える正の最小値
2.2250738585072014e-308  ←これが正の最小値 (正規化数24)
```

float の値は「(1+**仮数部**) × 2^{**指数部**}」に符号を付けた形式で表現されている. この場合の仮数部は, 2 進数表現した際の小数点以下の部分²⁵ であり, 仮数部のビット長は float_info.mant_dig から得られる. また, 指数部の最小値と最大値はそれぞれ float_info.min_exp, float_info.max_exp から得られる.

例. 仮数部のビット長, 指数部の最小値と最大値 (先の例の続き)

```
>>> sys.float_info.mant_dig  Enter    ←仮数部のビット長は
53                                ← 53 ビット
>>> sys.float_info.min_exp  Enter    ←指数部の最小値は
-1021
>>> sys.float_info.max_exp  Enter    ←指数部の最大値は
1024
```

float の値の 1.0 と $1.0 + \varepsilon$ を区別することができる最小の ε は float_info.epsilon から得られる²⁶.

例. 1.0 と $1.0 + \varepsilon$ を区別することができる最小の ε (先の例の続き)

```
>>> sys.float_info.epsilon  Enter    ←  $\varepsilon$ 
2.220446049250313e-16
```

■ 0.0 と -0.0 の違い

float 型では 0.0 と -0.0 は区別して扱われる.

例. 0.0 と -0.0

```
>>> a = 0.0; b = -0.0  Enter    ← 0.0 と -0.0
>>> print( 'a =', a, ', b =', b )  Enter    ←確認
a = 0.0 , b = -0.0          ← 0.0 と -0.0 は異なる
>>> a == b  Enter          ←しかし比較すると
True                        ←等しい
```

0.0 と -0.0 は比較演算子「==」で比較すると, 同じ値であると判定される.「==」に関しては, p.98「2.6.4.1 条件式」で説明する.

参考) math.atan2 関数における 0.0 と -0.0 の扱いの違い

Python の標準ライブラリである math モジュール²⁷ が提供する atan2 関数は 0.0 と -0.0 を明確に区別して取り扱う. この関数は 2 つの引数 y, x を取り, 座標の原点 (0,0) と (x,y) の偏角を求めるものである.

²³システムパラメータの取得や設定などに関するモジュール

²⁴float 型は「非正規化数」というさらに小さな値も扱えるが, ここでは正規化数についてのみ説明する.

²⁵ここで言う仮数部は, 真の仮数 $1.xxx\cdots$ の $xxx\cdots$ の部分であり, 先頭の 1. の部分は暗黙である. ただし, 符号以外のビットが全て 0 の場合, 先頭の 1. の部分は無いものと解釈され, 結果としての値は 0.0 もしくは -0.0 となる.

²⁶0.0 と $0.0 + \varepsilon$ の区別ではないことに注意すること.

²⁷後の「2.4.6.11 数学関数」(p.19) で解説する.

例. 原点 (0,0) と (-1,0) の偏角

```
>>> import math      [Enter]      ← math モジュールの読み込み
>>> math.atan2( 0.0, -1.0 ) [Enter] ← 原点と (-1.0, 0.0) の偏角は
3.141592653589793      ←  $\pi$  ラジアン
```

例. 原点 (0,0) と (-1,-0) の偏角 (先の例の続き)

```
>>> math.atan2( -0.0, -1.0 ) [Enter] ← 原点と (-1.0, -0.0) の偏角は
-3.141592653589793      ←  $-\pi$  ラジアン
```

この例からわかるように、座標 (-1,0) と (-1,-0) は同じに見えるが、それぞれ第 2 象限、第 3 象限と異なる領域にあり、atan2 関数はこれらを識別している。

※ 高い精度の（仮数部の桁数の大きい）浮動小数点数を扱う方法については後の「2.4.6.13 多倍長精度の浮動小数点数の扱い」（p.25）を参照のこと。

2.4.6.4 アンダースコア「_」を含む数値リテラルの記述

数値の記述の中にあるアンダースコア「_」は無視される。

例. アンダースコアを含む数値の記述

```
>>> 1_000_000 [Enter]      ← 記述例 1
1000000      ← 実際の値
>>> 2_345.67 [Enter]      ← 記述例 2
2345.67      ← 実際の値
>>> 9_8_7_6_5_4_3_2_1 [Enter] ← 記述例 3
987654321      ← 実際の値
```

ただし、小数点の記号と接続するとエラーが発生する。

例. 誤った形でアンダースコアを数値の中に記述した例

```
>>> 3._14 [Enter]      ← 誤った記述
File "<stdin>", line 1      ← エラーメッセージ
  3._14
    ^
SyntaxError: invalid syntax      ← 文法エラーを意味する
```

2.4.6.5 基数の指定：2進数, 8進数, 16進数

数を記述する際に表 3 に示すような接頭辞を付けることで 2 進法, 8 進法, 16 進法の表現ができる。

表 3: 基数 (n 進法) の接頭辞

接頭辞	表現	例
0b (ゼロビー)	2 進法	0b1011011 (= 91_{10})
0o (ゼロオー)	8 進法	0o1234567 (= 342391_{10})
0x (ゼロエックス)	16 進法	0x7b4f (= 31567_{10})

例. 2 進数, 8 進数, 16 進数

```
>>> 0b111 [Enter]      ← 2 進数表現の数値
7      ← 得られた値 (10 進数表現)
>>> 0o7777 [Enter]      ← 8 進数表現の数値
4095      ← 得られた値 (10 進数表現)
>>> 0xff [Enter]      ← 16 進数表現の数値
255      ← 得られた値 (10 進数表現)
```

この例のように、接頭辞を付けたものが直接に数値として扱われる。

2.4.6.6 複素数：complex 型

Python では複素数が扱える。この際、虚数単位は j で表す。

例. $(1+i)(1-i)$ の計算を Python で実行する

```
>>> (1+1j)*(1-1j)  Enter  
(2+0j)
```

このように**虚部** (j の係数) を明に記述する必要がある。すなわち、虚部が 1 や 0 であってもそれらを明に記述しなければならない。また計算の結果、虚部が 0 になっても $0j$ と表記される。

複素数の**実部** (実数の部分) が 0 の場合は実部の表示が省略される。また、実部、虚部ともに 0 の場合でも、値が complex 型の場合は $0j$ と表示される。

例. 実部、虚部が 0 となる例

```
>>> a = 0+2j  Enter  ←実部が 0, 虚部が 2 の複素数  
>>> a  Enter  ←内容確認  
2j  ←実部の表示が省略されている  
>>> a - 2j  Enter  ←更に虚部も 0 にしてみると  
0j  ← complex 型は虚部が必ず残る
```

複素数の**実部**と**虚部**は、プロパティ `real` と `imag` に保持されており、個々に取り出すことができる。

例. 実部、虚部の取得

```
>>> c = 1-3j  Enter  ←複素数の生成  
>>> c.real  Enter  ←実部の取り出し  
1.0  ←実部が得られる  
>>> c.imag  Enter  ←虚部の取り出し  
-3.0  ←虚部が得られる
```

複素数に対して `conjugate` メソッドを使用すると共役複素数が得られる。

例. 共役複素数 (先の例の続き)

```
>>> c.conjugate()  Enter  ←共役複素数の算出  
(1+3j)  ←共役複素数が得られている
```

■ complex コンストラクタ

`complex` コンストラクタで複素数を生成することもできる。

書き方: `complex(実部, 虚部)`

例. `complex` コンストラクタで複素数を生成

```
>>> complex(1,2)  Enter  ←実部が 1, 虚部が 2 の複素数を生成  
(1+2j)  ←結果表示
```

参考) 複素数は実部 a と虚部 b の和 $a+bj$ の形で表現されるため、複素数の計算の結果は基本的にこの形になる。

例. 複素数の除算 $(5+10i)/(3-4i)$

```
>>> (5+10j)/(3-4j)  Enter  ←複素数の除算  
(-1+2j)  ←結果は実部と虚部の和の形式
```

■ 複素数のノルム

複素数 $a+bi$ のノルム $\sqrt{a^2+b^2}$ は `abs` 関数で求めることができる。

例. $3+4i$ のノルム

```
>>> abs( 3+4j )  Enter  ← abs 関数による複素数のノルムの算出  
5.0  ←ノルムの値 (float 型で得られる)
```

`abs` 関数の引数に実数を与えると、その**絶対値**を返す。

例. -5 の絶対値を求める

```
>>> abs( -5 )  Enter  ← |-5| を求める  
5  ←結果表示
```

2.4.6.7 各種の数の型と数学的階層

数学的には、複素数 (\mathbb{C}) の一部として実数 (\mathbb{R}) があり、実数の一部として有理数 (\mathbb{Q}) がある。また、有理数の一部として整数 (\mathbb{Z}) がある。これまで、Python 言語で扱う整数 (int)、浮動小数点数 (float)、複素数 (complex) といった型について解説したが、数学的な階層として Python 言語の数値の型を分類する機能が numbers モジュールとして提供されている。これについては後の「4.30.3 数のクラス階層： numbers モジュール」(p.364) で解説する。

numbers モジュールを使用すると、後で解説する mpmath モジュール²⁸ が提供する数や、fractions モジュール²⁹ が提供する分数なども数学的階層に正しく位置づける形で分類することができる。

2.4.6.8 異なる型の数値同士の算術演算と型昇格規則

異なる型の数値同士の算術演算の結果として得られる値の型は、先に解説した、数値の型の階層関係（数値タワー³⁰）によって決まる。数値タワーの中では複素数 (\mathbb{C}) に関するクラス (complex など) の位置が最も高く、整数 (\mathbb{Z}) に関するクラス (int など) の位置が最も低い。また、異なる型の数値同士の算術演算の結果は、数値タワー上の高い方の型となる（型昇格規則）。

例. 異なる型の数値同士の加算

```
>>> 1 + 2.0  Enter  ← int と float の加算では
3.0          ←結果は float となる
>>> 2.0 + (3+7j) Enter  ← float と complex の加算では
(5+7j)       ←結果は complex となる
```

2.4.6.9 最大値, 最小値

関数 max, min の引数に値の列を与えることで、それぞれ最大値, 最小値を求めることができる。

例. 最大値と最小値の算出

```
>>> max( 1, 2, 3, 4, 3, 2 ) Enter  ←引数の中から最大の値を選び出す
4      ←結果表示
>>> min( 1, 2, 3, 4, 3, 2 ) Enter  ←引数の中から最小の値を選び出す
1      ←結果表示
```

max, min の引数にはリストやタプル³¹ を与えることもできる。

例. 最大値と最小値の算出（リストの中から選出）

```
>>> a = [1,2,3,4,3,2] Enter  ←値の列のリストを作成
>>> max( a ) Enter  ←リスト a の中から最大の値を選び出す
4      ←結果表示
>>> min( a ) Enter  ←リスト a の中から最小の値を選び出す
1      ←結果表示
```

参考) 複数の値の合計値を求める関数として sum がある。これに関しては「2.5.1.8 要素の合計： sum 関数」(p.65) で解説する。

2.4.6.10 浮動小数点数の誤差と丸め

浮動小数点数の計算の結果は誤差を含む。

例. 浮動小数点数の計算における誤差

```
>>> 0.1 + 0.2  Enter  ←加算
0.30000000000000004  ←誤差を含んだ計算結果
```

数値は実際には（計算機内では）2 進数の形式で表現される。また、10 進数表現において「0.1」は有限の桁数で表現できるが、これを 2 進数で表す場合は循環小数となる。従って、そのような値を計算に用いる場合は、精度の上限（扱える桁数の上限³²）によって計算結果の桁数が制限され、これに起因する計算の誤差が生じる。

²⁸ 「2.4.6.13 多倍長精度の浮動小数点数の扱い」(p.25) で解説する。

²⁹ 「2.4.6.14 分数の扱い」(p.28) で解説する。

³⁰ Lisp 言語の 1 方言である Scheme (R3RS) に関する文書で明確化された概念。Python では PEP 3141 で言及されている。

³¹ リスト, タプルに関しては「2.5 データ構造」(p.57) で解説する。

³² float の有効桁数は約 15 桁程度である。

値によっては 2 進数表現の際に循環小数とならないこともあり、その場合は計算結果に誤差が生じない。

例. 誤差のない浮動小数点数の計算

```
>>> 0.5 + 0.25  Enter  ←加算
0.75             ←誤差が見られない
```

この例の場合は $0.5_{(10)} = 0.1_{(2)}$, $0.25_{(10)} = 0.01_{(2)}$ である³³ ので、10 進数、2 進数の両方の表現において有限の桁数となり（循環小数とはならず）計算結果に誤差が生じない。

参考) 正確な 10 進演算を行うための特別な方法については、後の「2.4.6.16 正確な 10 進演算」(p.31) で解説する。

指定した桁数に値を丸めるには round 関数を使用する。この関数は浮動小数点数の小数点以下の桁を丸める（四捨五入する）。

書き方: round(値, 小数点以下の桁数)

この関数は、引数に与えた「値」の小数点以下を「小数点以下の桁数」に丸めた結果を返す。

例. round 関数

```
>>> 1.0 / 3.0  Enter  ← 1 ÷ 3 の計算（丸めなし）
0.3333333333333333  ←結果の表示
>>> round( 1.0 / 3.0, 2 )  Enter  ← 1 ÷ 3 の計算（小数点以下 2 桁にする丸め）
0.33                  ←結果の表示
>>> round( 123456789.123, -2 )  Enter  ←桁数指定に負の値を指定
123456800.0           ←結果の表示
```

この例の様に、丸めの桁数に負の値を指定することができ、その場合は「整数部の末尾の丸め」となる。

round 関数の第 2 引数を省略すると小数点以下が丸めの対象となり、結果を整数 (int 型) として返す。

例. 第 2 引数の省略

```
>>> round( 3.4 )  Enter  ←この場合は
3                  ←小数点以下を切り下げた整数
>>> round( 3.6 )  Enter  ←この場合は
4                  ←小数点以下を切り上げた整数
```

round 関数の処理は厳密には四捨五入ではなく偶数丸め³⁴ (Banker's Rounding または Round Half To Even) である。これは、切り下げ／切り上げ対象部分が唯 1 桁の「5」である場合、処理結果の末尾の桁が偶数になるように丸める処理である。(次の例参照)

例. 偶数丸め (その 1)

```
>>> round( 1.25, 1 )  Enter  ←この場合は末尾が「5」なので
1.2                    ←右端が偶数になる
>>> round( 1.35, 1 )  Enter  ←この場合も末尾が「5」なので
1.4                    ←右端が偶数になる
>>> round( 2.5 )  Enter  ←この場合は
2                      ←小数点以下が切り下げられて偶数となる
>>> round( 3.5 )  Enter  ←この場合は
4                      ←小数点以下が切り上げられて偶数となる
```

例. 偶数丸め (その 2)

```
>>> round( 25, -1 )  Enter  ←この場合は整数部の末尾が「5」なので
20                      ←右から 2 桁目が偶数となる
>>> round( 35, -1 )  Enter  ←この場合も整数部の末尾が「5」なので
40                      ←右から 2 桁目が偶数となる
```

2.4.6.11 数学関数

次のようにして math モジュールを読み込むことで、各種の数学関数が使用できる。

³³参考: 巻末付録「I.5 浮動小数点数と 2 進数の間の変換」(p.462)

³⁴統計処理や金融関連の計算処理で採用されている。

例. math モジュールの読み込み

```
>>> import math
```

各種関数は「math.」の接頭辞を付けて呼び出す。例えば正弦関数の値を求めるには次のようにする、

例. 正弦関数 (sin) の使用

```
>>> math.sin(math.pi/2)
1.0
```

表 4 に math モジュールが提供する数学関数や定数の一部を挙げる。表 4 に挙げたもの以外にも多くの関数が提供されている。詳しくは Python の公式インターネットサイトを参照のこと。

表 4: 使用できる数学関数, 定数の一部

関数	説明	関数	説明
<code>sqrt(x)</code>	x の平方根 \sqrt{x}	<code>pow(x,y)</code>	x の y 乗 x^y
<code>exp(x)</code>	x の指数関数 e^x	<code>log(x,y)</code>	x の対数関数 $\log_y x$ ※ 1
<code>log2(x)</code>	x の対数関数 $\log_2 x$	<code>log10(x)</code>	x の対数関数 $\log_{10} x$
<code>sin(x)</code>	x の正弦関数 $\sin(x)$	<code>cos(x)</code>	x の余弦関数 $\cos(x)$
<code>tan(x)</code>	x の正接関数 $\tan(x)$	<code>asin(x)</code>	x の逆正弦関数 $\sin^{-1}(x)$
<code>acos(x)</code>	x の逆余弦関数 $\cos^{-1}(x)$	<code>atan(x)</code>	x の逆正接関数 $\tan^{-1}(x)$
<code>sinh(x)</code>	x の双曲線正弦関数 $\sinh(x)$	<code>cosh(x)</code>	x の双曲線余弦関数 $\cosh(x)$
<code>tanh(x)</code>	x の双曲線正接関数 $\tanh(x)$	<code>asinh(x)</code>	x の逆双曲線正弦関数 $\sinh^{-1}(x)$
<code>acosh(x)</code>	x の逆双曲線余弦関数 $\cosh^{-1}(x)$	<code>atanh(x)</code>	x の逆双曲線正接関数 $\tanh^{-1}(x)$
<code>pi</code>	円周率 π	<code>e</code>	ネイピア数 e
<code>atan2(y,x)</code>	座標原点と点 (x,y) の偏角 (弧度法) を返す関数		
<code>floor(x)</code>	x を小さい方の整数値として丸める	<code>ceil(x)</code>	x を大きい方の整数値として丸める
<code>trunc(x)</code>	x の小数点以下を切り取る		

※ 1 y を省略した場合は自然対数

例. $n!$ を求める関数 factorial

```
>>> import math
>>> math.factorial( 30 )
265252859812191058636308480000000
```

■ 最大公約数 (gcd) と最小公倍数 (lcm) について

math モジュールは**最大公約数**を求める関数 gcd を提供するが、Python の版によって仕様が異なることに注意しなければならない。(次の例参照)

例. Python3.8 における gcd 関数

```
>>> import math
>>> math.gcd(12,18)
6
>>> math.gcd(12,18,24)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: gcd expected 2 arguments, got 3
```

この例からわかるように、Python3.8 の gcd 関数は 2 つ整数の最大公約数を求めることはできるが、3 つ以上の整数については計算することができない。これに対して、Python3.9 以降の版の gcd 関数では 3 つ以上の整数について計算することができる。(次の例参照)

例. Python3.9 における gcd 関数

```
>>> import math  ←モジュールの読み込み
>>> math.gcd(12,18)  ← 12 と 18 の最大公約数を求める
6      ←計算結果
>>> math.gcd(12,18,24)  ← 12 と 18 と 24 の最大公約数を求める
6      ←計算結果
```

Python3.9 以降では、math モジュールは最小公倍数を求める関数 lcm を提供する。(3.8 以前では未提供)

例. Python3.9 における lcm 関数

```
>>> import math  ←モジュールの読み込み
>>> math.lcm(4,6)  ← 4 と 6 の最小公倍数を求める
12      ←計算結果
```

例. Python3.8 では lcm 関数は未提供

```
>>> import math  ←モジュールの読み込み
>>> math.lcm(4,6)  ← lcm 関数を呼び出そうとすると…
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
AttributeError: module 'math' has no attribute 'lcm'
```

■ pow 関数

math モジュールは pow 関数を提供するが、これとは別に、Python の組み込み関数にも同名の関数がある。

例. pow 関数の比較

```
>>> pow(2,3)  ←組み込み関数の pow による 23
8      ←整数の 8
>>> import math  ←モジュールの読み込み
>>> math.pow(2,3)  ← math.pow による 23
8.0    ← float の 8.0
```

このように、与える数によって計算結果の型が異なることがある。また、複素数の扱いは明確に異なる。(次の例)

例. pow 関数の複素数の扱い (先の例の続き)

```
>>> pow(-2+0j,0.5)  ←組み込み関数の pow による -20.5
(8.659560562354934e-17+1.4142135623730951j) ←結果は複素数
>>> math.pow(-2+0j,0.5)  ← math.pow による -20.5 の試み
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    math.pow(-2+0j,0.5)
    ~~~~~
TypeError: must be real number, not complex
```

このように math.pow は複素数の計算には対応していない。複素数の平方根を計算するには、次に解説する cmath モジュールの関数を使用する。

■ 複素数を扱う関数

math モジュールが提供する関数群は実数に対するものである。複素数を扱う関数群は cmath モジュールが提供しており、表 4 に挙げたものと共通の関数名のことが多い。詳しくは Python の公式ドキュメント (公式インターネットサイトなど) を参照のこと。

例. math モジュール, cmath モジュールそれぞれの sqrt 関数の違い

```
>>> import math      [Enter]      ← math モジュールの読み込み
>>> math.sqrt(-2)     [Enter]      ← math モジュールの関数で  $\sqrt{-2}$  を算出する試み
Traceback (most recent call last):      ← math.sqrt が複素数を扱えないため
  File "<stdin>", line 1, in <module>     エラーが発生する
ValueError: math domain error

>>> import cmath      [Enter]      ← cmath モジュールの読み込み
>>> cmath.sqrt(-2)     [Enter]      ← cmath モジュールの関数で  $\sqrt{-2}$  を算出する試み
1.4142135623730951j      ← 計算結果が複素数として得られる
```

2.4.6.12 特殊な値: inf, nan

数値として扱われる特殊な記号として inf と nan がある. inf は, あらゆる数値より大きい値 (無限大) として定義された形式的な記号である. (次の例参照)

例. 形式的な無限大記号の扱い

```
>>> import math      [Enter]      ← math モジュールの読み込み
>>> math.inf          [Enter]      ← 形式的な無限大
inf                  ← 形式上の表示

>>> a = math.inf       [Enter]      ← 変数に代入することもできる
>>> a > 100**10000     [Enter]      ← 非常に大きな数値  $100^{10000}$  との比較
True                 ← inf はあらゆる数値よりも大きい
```

この例で使用している比較演算子「>」に関しては, p.98 「2.6.4.1 条件式」で説明する. また, True という値は真か偽かを表す値「真理値」であり, これについては p.50 「2.4.8 真理値」で説明する.

inf は math モジュールに依らず使用することができる. (次の例参照)

例. math モジュールに依らない inf の生成

```
>>> a = float('inf')  [Enter]      ← inf の生成
>>> a                  [Enter]      ← 内容確認
inf                  ← inf が生成されている

>>> type(a)            [Enter]      ← 型の検査 (p.52 「2.4.12 型の検査」で説明する)
<class 'float'>      ← 浮動小数点数 (float) の型である
```

inf は負 (マイナス) の符号を取り得る. (次の例参照)

例. 負の inf (先の例の続き)

```
>>> b = -1*a          [Enter]      ← (-1) を掛けて負の inf をつくる
>>> b                  [Enter]      ← 内容確認
-inf                 ← 負の inf が生成されている

>>> b < -(100**10000)  [Enter]      ← 非常に絶対値の大きな負の数値  $-(100^{10000})$  との比較
True                 ← -inf はあらゆる数値よりも小さい
```

inf 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, 値として数値計算に用いるべきではない. (次の例参照)

例. inf を計算に用いる試み (先の例の続き)

```
>>> a + a              [Enter]      ← 加算の試み
inf                  ← 計算結果

>>> a * a              [Enter]      ← 乗算の試み
inf                  ← 計算結果

>>> a - a              [Enter]      ← 減算の試み
nan                  ← 計算結果 (数ではない)

>>> a / a              [Enter]      ← 除算の試み
nan                  ← 計算結果 (数ではない)
```

この例からわかるように減算と除算の結果は解釈できない。これは**非数**であり、nan (not a number) という記号で形式的に表される。

注意) inf, nan の型は float である。

■ nan (非数) の性質

nan はどのような数学的性質も持たない、あくまでも「計算不可能な処理の結果」を表す便宜的なオブジェクトである。次に示す例の結果は一見して不可解であるが、nan が数学的処理に全く一貫性が無いことを示している。

例. nan を用いた処理

```
>>> import math      Enter    ←モジュールの読み込み
>>> a = math.nan      Enter    ← nan を変数 a に代入
>>> a                  Enter    ←確認
nan
>>> a < 1              Enter    ← nan は 1 より小さいか？
False                  ←偽
>>> a >= 1             Enter    ← nan は 1 以上か？
False                  ←偽
>>> a == a             Enter    ← nan は nan に等しいか？
False                  ←偽
```

比較演算子「>=」「==」に関しては、p.98 「2.6.4.1 条件式」で説明する。

■ 特殊な値かどうかの判定

math モジュールは 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。

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

例. 無限大の判定 (先の例の続き)

```
>>> math.isinf( a )    Enter    ← a (inf) を判定
True                  ← a は正の無限大 inf である
>>> math.isinf( -a )   Enter    ← -a (-inf) を判定
True                  ← -a は負の無限大 -inf である
>>> math.isinf( 2 )    Enter    ← 2 を判定
False                 ← 2 は有限の数値である
>>> math.isinf( a/a )   Enter    ← a/a (nan) を判定
False                 ← nan は無限大ではない
```

例. 非数 (nan) の判定 (先の例の続き)

```
>>> math.isnan( a/a )   Enter    ← a/a (nan) を判定
True                  ←非数 (nan) である
>>> math.isnan( a )     Enter    ← a (inf) を判定
False                 ← inf は nan ではない
>>> math.isnan( 2 )     Enter    ← 2 を判定
False                 ← 2 は nan ではない
```

例. 有限な数値の判定 (先の例の続き)

```
>>> math.isfinite( 2 )   Enter    ← 2 を判定
True                  ←有限な数値である
>>> math.isfinite( a )   Enter    ← a (inf) を判定
False                 ← inf は有限な数値ではない
>>> math.isfinite( a/a ) Enter    ← a/a (nan) を判定
False                 ← nan は有限な数値ではない
```


■ inf, nan の IEEE-754 での取り決め

inf, nan は IEEE-754 で次のように取り決められており, Python もそれに準拠している.

inf : 指数部の全ビットは 1, 仮数部の全ビットは 0

nan : 指数部の全ビットは 1, 仮数部は非ゼロ³⁵, 符号部は関知しない

【参考】 inf, nan のビットパターンを調べる.

Python のオブジェクトとバイト列³⁶ の相互変換機能を提供する struct モジュール³⁷ を使って、特殊な値のビットパターンを調べる。

例. math.inf のビットパターン

[illegible]

ビットパターンを文字列³⁸ として作成している。(文字列を連結する join メソッドについては後の「リストの要素の連結」(p.42)で解説する)

符号ビット（第 0 ビット）が 0、指数部（第 1～11 ビット）の全ビットが 1、仮数部の全ビット（第 12～63 ビット）が 0 となっていることがわかる。

例. -math.inf のビットパターン (先の例の続き)

[illegible]

符号ビット（第0ビット）が1となっており負の値（ $-\infty$ ）となっていることがわかる。（指数部と仮数部は先の例と同じ）

例. math.nan のビットパターン (先の例の続き)

```
>>> b = struct.pack(>d', math.nan) Enter ← math.nan をバイト列に変換
>>> bits = ''.join(f'byte:08b' for byte in b) Enter ←ビットパターンの文字列作成
>>> print(bits) Enter ←確認
011111111111100000000000000000000000000000000000000000000000000000000000
```

指数部（第 1～11 ビット）の全ビットが 1、仮数部が非ゼロ（第 12 ビットが 1）となっていることがわかる。

ここで示した例では join メソッドの引数に特殊な引数を与えているが、これに関しては、後の「引数部分に内包表記を与えた場合の動作」(p.145)で解説する。また「内包表記」に関しては後の「2.6.1.7 for を使ったデータ構造の生成(要素の内包表記)」(p.92)のところで解説する。

³⁵ 仮数部のビットパターンによって、更に qNaN, sNaN に分類されるが、本書では解説を割愛する。

³⁶ 「2.4.15 バイト列」(p.56)で解説する.

³⁷ 「4.11 バイナリデータの作成と展開：struct モジュール」(p.313)で解説する。

³⁸ 「2.4.7 文字列」(p.38)で解説する.

2.4.6.13 多倍長精度の浮動小数点数の扱い

<http://mpmath.org/> で公開されている mpmath ライブラリ³⁹ を使用すると、IEEE-754 の倍精度浮動小数点数の精度に制限されない**任意の精度**による浮動小数点数の演算が可能となる。mpmath を使用するには必要なものを次のようにして読み込んでおく。

```
from mpmath import mp
```

mpmath ライブラリには math モジュールで提供されているものと類似の関数が多数提供されており、表 4 のような関数に接頭辞 'mp.' を付ける⁴⁰ ことでその関数の値を求めることができることが多い。ただし math モジュールとの厳密な違いには注意すること。mpmath の使用方法の詳細に関しては先のサイトを参照のこと。

■ 演算精度の設定（仮数部の桁数の設定）

mp の dps プロパティに整数値を設定することで、計算精度（仮数部の桁数）を設定することができる。

例. 円周率を 200 桁求める

```
>>> from mpmath import mp      Enter      ← mpmath ライブラリの読み込み
>>> mp.dps = 200                Enter      ← 演算精度の設定
>>> print( mp.pi )             Enter      ← 円周率の算出
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280
348253421170679821480865132823066470938446095505822317253594081284811174502841027019385
21105559644622948954930382      ← 計算結果
```

注 1) この例では演算精度に 200 桁を指定しているが結果的に 199 桁の仮数部が得られている。従って、結果的に得られる値の桁数の上限を mp.dps に設定すると考えるべきである。

注 2) mpmath ライブラリの関数が返す値の型は、通常の浮動小数点数 float とは異なる mpf オブジェクト (mpmath.ctx_mp_python.mpf) であることに注意すること。

■ 値の生成

mpmath が扱う数値は mpf クラスのインスタンスであり、mpf コンストラクタに初期値を与えて生成⁴¹ する。

例. 値の生成（mpf オブジェクトの生成）

```
>>> from mpmath import mp      Enter      ← mpmath ライブラリの読み込み
>>> mp.dps = 72                Enter      ← 演算精度の設定
>>> a = mp.mpf('0.0000000001')  Enter      ← 10-10 のオブジェクト生成して a に設定
>>> b = mp.mpf('1.0e10')        Enter      ← 1010 のオブジェクトを生成して b に設定
>>> a*b                         Enter      ← 積の算出
mpf('1.0')                     ← 計算結果
```

a*b の計算結果が mpf オブジェクトとして得られていることがわかる。mpf オブジェクトは print 関数により自然な形で出力することができる。

例. print 関数による mpf オブジェクトの出力（先の例の続き）

```
>>> print( a*b )               Enter
1.0                             ← 自然な形の出力
```

mpf コンストラクタの引数には様々な型のオブジェクト (int, float, str) を与えることができる。特に仮数部の桁の長い浮動小数点数を与える場合は、文字列 (str) として与えることができる。文字列として浮動小数点数を記述する場合は先の例の様に**指数表記**が使える。

mpf オブジェクト a の値を int, float, str などの型の値に変換するには次のようにする。

```
整数に変換      : int( a )
浮動小数点数に変換 : float( a )
文字列に変換    : str( a )
```

³⁹PSF 版 Python での導入方法に関しては、巻末付録「A.4 PIP によるライブラリ管理」(p.404) を参照のこと。Anaconda の場合は既にインストールされていることが多い。Anaconda でのパッケージ管理は Anaconda Navigator や conda コマンドで行う。

⁴⁰ライブラリの読み込みと接頭辞の扱いについては若干の注意が求められることがある。詳しくは巻末付録「D ライブラリの取り扱いについて」(p.433) を参照のこと。

⁴¹クラス、インスタンス、コンストラクタに関することは後の「2.9 オブジェクト指向プログラミング」(p.154) で解説する。

■ 複素数

mpmath では複素数を扱うこともできる。虚数単位は Python 標準の表記と同じ「j」である。

例. -2 の平方根を求める

```
>>> from mpmath import mp Enter      ← mpmath ライブラリの読み込み
>>> n = mp.mpf( -2.0 ) Enter      ← mpmath の -2 を生成
>>> a = mp.sqrt( n ) Enter      ← 平方根の算出
>>> print( a ) Enter      ← 内容確認
(0.0 + 1.4142135623731j)      ← 複素数として得られる ( $0 + \sqrt{2}i$  を意味する)
>>> a Enter      ← 内部表現の確認
mpc(real='0.0', imag='1.4142135623730951') ← mpc オブジェクトとなっている
```

この例でわかるように、複素数は mpmath の mpc オブジェクトとして扱われる。実部、虚部はプロパティ real, imag にそれぞれ保持されている。

例. 実部、虚部の取り出し（先の例の続き）

```
>>> a.real Enter      ← 実部の取り出し
mpf('0.0')      ← 実部
>>> a.imag Enter      ← 虚部の取り出し
mpf('1.4142135623730951') ← 虚部
```

複素数を明に生成するには mpc コンストラクタを用いて

```
mp.mpc( 実部, 虚部 )
```

とする。

例. 虚数単位 i の 2 乗を求める（先の例の続き）

```
>>> n = mp.mpc( 0, 1 ) Enter      ← mpmath の虚数単位を生成
>>> a = n**2 Enter      ← 2 乗の算出
>>> print( a ) Enter      ← 内容確認
(-1.0 + 0.0j)      ← 計算結果は複素数として得られる
```

mpc コンストラクタには Python 標準の複素数（complex 型の値）を初期値として与えても良い。

例. complex 型の値を mpc オブジェクトに変換する（先の例の続き）

```
>>> mp.mpc( 2+3j ) Enter      ← complex 型の 2+3j を mpc オブジェクトにする
mpc(real='2.0', imag='3.0')      ← 得られた mpc オブジェクト
```

■ 特殊な値：無限大, 非数

mpmath パッケージも独自に無限大と非数を定義している。（次の例参照）

例. mpmath の無限大と非数

```
>>> from mpmath import mp Enter      ← mpmath ライブラリの読み込み
>>> a = mp.inf Enter      ← 無限大の生成
>>> a Enter      ← 内容確認
mpf('+inf')      ← mpmath での無限大
>>> -a Enter      ← 負の無限大の算出
mpf('-inf')      ← mpmath での負の無限大
>>> a + a Enter      ← 加算の試み
mpf('+inf')      ← 計算結果
>>> a - a Enter      ← 減算の試み
mpf('nan')      ← 計算結果（数ではない）
```

■ 符号, 仮数部, 指数部

mpf オブジェクトとして表される数の符号（+/-）を調べるには sign メソッドを使用する。

例. 符号を調べる

```
>>> from mpmath import mp  ←ライブラリの読み込み
>>> mp.dps = 25  ←演算精度の設定
>>> n1 = mp.mpf( '0.3' )  ← 0.3 の
>>> mp.sign( n1 )  符号を調べる
mpf('1.0') ← 1.0 : 正
>>> n2 = mp.mpf( '0' )  ← 0 の
>>> mp.sign( n2 )  符号を調べる
mpf('0.0') ← 0.0 : ゼロ
>>> n3 = mp.mpf( '-0.3' )  ← -0.3 の
>>> mp.sign( n3 )  符号を調べる
mpf('-1.0') ← -1.0 : 負
```

この例のように「`mp.sign(調査対象の値)`」として評価すると、正の場合は 1.0、ゼロの場合は 0、負の場合は -1.0 が得られる。(戻り値は `mpf` オブジェクト)

`mpf` オブジェクトは**仮数部** (mantissa) と**指数部** (exponent) から成り、それぞれ `man` プロパティ、`exp` プロパティから `int` 型で得られる。

例. 仮数部、指数部を調べる (先の例の続き)

```
>>> n1.man  ←仮数部を調べる
23211375736600880154358579 ←仮数部
>>> n1.exp  ←指数部を調べる
-86 ←指数部
```

この例から 0.3 が $23211375736600880154358579 \times 2^{-86}$ であることがわかる。また、仮数部、指数部の値は元の数の正負に依らない。

例. `mpf('-0.3')` の仮数部と指数部 (先の例の続き)

```
>>> n3.man  ←仮数部を調べる
23211375736600880154358579 ←仮数部
>>> n3.exp  ←指数部を調べる
-86 ←指数部
```

先の例と同じ結果となっていることがわかる。

【参考】 浮動小数点数の 2 進数表現

`mpf` オブジェクトの `man`、`exp` プロパティを参照することで、浮動小数点数の 2 進数表現を得ることができる。

例. 0.3 の 2 進数表現 (先の例の続き)

```
>>> bin(n1.man)  ←仮数部を 2 進数に変換
'0b1001100110011001100110011001100110011001100110011001100110011001100110011001100110011'
```

この結果から、0.3 の 2 進数表現 (近似表現) が

$$10011_{(2)} \times 2^{-86}$$

であることがわかる。

【参考】 `mpmath` とは別の多倍長精度数値演算ライブラリ: `gmpy2`

`mpmath` は Python 言語で構築されたライブラリであるが、同様の機能を持つ、別のライブラリとして `gmpy2` が有名である。`gmpy2` は GNU GMP/MPFR (C/C++ 言語用の多倍長精度数値演算ライブラリ) を Python から利用できるようにしたものである。

これに関しては本書では解説しない⁴² が、公式インターネットサイト⁴³ などで情報が公開されている。

⁴² 拙書「Python3 ライブラリブック」で解説しています。

⁴³ <https://github.com/aleaxit/gmpy>, <https://pypi.org/project/gmpy2/>

2.4.6.14 分数の扱い

Python 処理系に標準的に添付されている fractions モジュールを用いると分数を扱うことができる。このモジュールは次のようにして Python 処理系に読み込む。

```
from fractions import Fraction
```

これにより、分数を扱うための Fraction クラスが利用できる。

例. 分数を生成して計算する例

```
>>> from fractions import Fraction  [Enter]  ← fractions モジュールの Fraction クラスを読み込む
>>> a = Fraction(2,3)  [Enter]  ← 分数 2/3 を生成して a に代入
>>> b = Fraction(3,4)  [Enter]  ← 分数 3/4 を生成して b に代入
>>> c = a + b  [Enter]  ← 2つの分数を加算
>>> c  [Enter]  ← 内容確認
Fraction(17, 12)  ← 計算結果
>>> print( c )  [Enter]  ← print 関数で出力
17/12  ← 整形表示されている
```

この例のように Fraction コンストラクタの第一引数に分子を、第二引数に分母を与えることで分数オブジェクトが生成される。分数オブジェクトは整数 (int) や浮動小数点数 (float) とは別のクラスのオブジェクトであり、約分され既約な形で保持される。

浮動小数点数 (float 型) や文字列 (str 型) から直接的に分数を作成することもできる。

例. 浮動小数点数から分数に変換する例 (先の例の続き)

```
>>> d = 1.23456  [Enter]  ← 浮動小数点数 1.23456 を用意
>>> Fraction(d)  [Enter]  ← Fraction への変換
Fraction(694995494495815, 562949953421312)  ← 変換結果
```

この例から、 $1.23456 \approx \frac{694995494495815}{562949953421312}$ であることがわかる。

また同様のことが、Fraction クラスのメソッド from_float によっても可能である。(次の例)

例. from_float メソッドによる変換 (先の例の続き)

```
>>> Fraction.from_float( d )  [Enter]  ← from_float メソッドによる変換
Fraction(694995494495815, 562949953421312)  ← 変換結果
```

例. 文字列表現の分数を Fraction にする (先の例の続き)

```
>>> Fraction('3/5')  [Enter]  ← Fraction の引数に文字列表現の分数を与える
Fraction(3, 5)  ← 変換結果
```

分数から浮動小数点数、あるいは文字列に変換することもできる。

例. 分数を浮動小数点数や文字列に変換する例 (先の例の続き)

```
>>> float( c )  [Enter]  ← 分数を浮動小数点数に変換する
1.4166666666666667  ← 変換結果
>>> str( c )  [Enter]  ← 分数を文字列に変換する
'17/12'  ← 変換結果
```

分数は int や float との演算が可能である。(次の例参照)

例. 分数と整数、浮動小数点数との計算 (先の例の続き)

```
>>> print( 1 + c )  [Enter]  ← 分数と整数の加算
29/12  ← 計算結果 (分数として得られる)
>>> print( 0.5 + c )  [Enter]  ← 分数と浮動小数点数の加算
1.9166666666666667  ← 計算結果 (浮動小数点数として得られる)
```

分数と整数の演算では分数の形で、分数と浮動小数点数の演算では浮動小数点数の形で結果が得られる。

■ 分母の大きさの制限

分数 (Fraction オブジェクト) に対して limit_denominator メソッドを使うと分母の大きさを制限した形で近似した分数を得ることができ、様々な場面で応用ができる。例えば 2.2 という値を分数で表現すると 11/5 になると思

われるが、実際に float 型の 2.2 を Fraction に変換すると次のようになる。

例. float の 2.2 を Fraction に変換する

```
>>> f = Fraction( 2.2 )  Enter  ← 2.2 を Fraction に変換
>>> print( f )  Enter  ←表示して確認
2476979795053773/1125899906842624  ←変換結果
```

分母、分子ともに非常に大きな値となっていることがわかる。これは、float 型の 2.2 が計算機の内部では循環小数 $10.00110011001100110011\dots_{(2)}$ となっていることが原因である。次に分母が 10 を超えない形に近似した分数を求める例を示す。

例. 分母が 10 を超えないように近似（先の例の続き）

```
>>> print( f.limit_denominator(10) )  Enter  ←分母の上限を 10 にした形の近似値を求める
11/5  ←近似値の分数
```

この例のように limit_denominator の引数に分母の上限値を与えて実行する。

■ 分母、分子の取り出し

Fraction オブジェクトの分子と分母は numerator プロパティと denominator プロパティがそれぞれ保持している。

例. Fraction の分子と分母

```
>>> f = Fraction(13,17)  Enter  ← Fraction オブジェクトの生成
>>> print( f )  ←内容確認
13/17  ←結果表示
>>> f.numerator  Enter  ←分子の参照
13  ←分子
>>> f.denominator  Enter  ←分母の参照
17  ←分母
```

参考) 浮動小数点数に対して as_integer_ratio メソッドを実行すると、それを表現する分数の分子と分母のペアが得られる。

例. 分子と分母のペアの取得

```
>>> (3.5).as_integer_ratio()  Enter  ← 3.5 を分数と見た場合の分子と分母の取得
(7, 2)  ←結果表示
```

これは、3.5 を $7/2$ と見て、分子と分母である 7 と 2 のペアを **タプル**⁴⁴ という括弧で括られたデータ構造として取得している例である。

2.4.6.15 除算における商と剰余に関する事柄

Python の整数除算と剰余は独特であることに注意すること。

例. Python における整数除算

```
>>> 10 // 3  Enter  ← 10 ÷ 3 の整数除算の商
3
>>> -10 // 3  Enter  ← -10 ÷ 3 の整数除算の商
-4  ←この結果に注意！
```

このように、Python では整数除算に演算子「//」を用いるが、 $-10 // 3$ の結果は意外な印象を与える。例えば、C 言語では、整数の -10 を整数の 3 で除算すると結果は -3 となる。また、JavaScript 言語でも同様の計算を

```
parseInt( -10 / 3 )
```

として行くと、結果は -3 (C 言語の場合と同じ) となる。これは、C 言語や JavaScript 言語が、除算の結果の小数点以下を単純に切り落としているからである。それに対して Python は除算の結果を、小さい方の整数に丸めて (math.floor 関数の処理) いる。Python の「//」演算子と同等の計算を敢えて math.floor 関数で実行する例を次に示す。

⁴⁴タプルに関しては「2.5.2 タプル」(p.68) で説明する。

例. Python における整数除算 (その 2)

```
>>> import math  ← math モジュールの読み込み
>>> math.floor( 10 / 3 )  ← 「//」 演算子と同等の計算
3
>>> math.floor( -10 / 3 )  ← 「//」 演算子と同等の計算
-4
```

また, C 言語や JavaScript 言語と同等の整数除算を実現するには次のようにする.

例. C や JavaScript と同等の整数除算 (先の例の続き)

```
>>> math.trunc( 10 / 3 )  ← C 言語と同等の整数除算
3
>>> math.trunc( -10 / 3 )  ← C 言語と同等の整数除算
-3
```

以上のことと併せて, **剰余**についても理解しておかねばならない. C 言語や JavaScript 言語における剰余の演算子「%」と Python のそれ⁴⁵は異なる. C 言語や JavaScript 言語において $10 \% 3$ の値は Python の場合と同じく 1 となるが, $-10 \% 3$ の値は C, JavaScript では -1, Python では 2 となる.

例. Python での剰余

```
>>> 10 % 3  ← 10 ÷ 3 の剰余 (余り) は
1 ← Python, C, JavaScript で同じ結果
>>> -10 % 3  ← -10 ÷ 3 の剰余 (余り) は
2 ← Python は C, JavaScript の場合と異なる
```

これは, 剰余を求める際の商の仕様の違いに由来する.

商と剰余を同時に求める `divmod` 関数についても, Python の整数除算の性質が影響する.

例. `divmod` 関数

```
>>> divmod(10,3) 
(3, 1) ← 商と剰余
>>> divmod(-10,3) 
(-4, 2) ← 商と剰余
```

参考) 用途によっては, 除算の結果 (商) を大きい方の整数に丸めた (`math.ceil` 関数) ものとして剰余を求める考え方もある.

課題. 正負様々な浮動小数点数における剰余の計算結果を「%」演算子を用いて確かめよ. また, C 言語や JavaScript 言語における結果とも比較せよ.

⁴⁵Python の % 演算子は, 数学の mod 演算 (モジュロ演算) に基づいている.

2.4.6.16 正確な 10 進演算

標準ライブラリである decimal モジュールを用いると、正確な 10 進演算ができる。このライブラリは、10 進演算のための特別な数値型 Decimal を提供し、これに基づいた計算処理を行う。このライブラリの使用に当たっては、

```
from decimal import Decimal, getcontext
```

として、必要なものを読み込む。

先の「2.4.6.10 浮動小数点数の誤差と丸め」(p.18) で解説した通り、float 型の数値はシステム内部で 2 進数で表現 (IEEE-754) されており、このことに起因する計算の誤差が発生する。(次の例)

例. float 型の計算誤差 (再掲載)

```
>>> 0.1 + 0.2      ←このような単純な計算でも  
0.30000000000000004    ←明確に誤差が発生する
```

同様の計算を decimal モジュールを用いて実行する例を示す。

例. decimal モジュールの Decimal 型の数値

```
>>> from decimal import Decimal, getcontext     ←モジュールの読み込み  
>>> d1 = Decimal('0.1')     ← Decimal 型の 0.1  
>>> d2 = Decimal('0.2')     ← Decimal 型の 0.2  
>>> d1     ← d1 の確認  
Decimal('0.1')    ←データができています  
>>> d2     ← d2 の確認  
Decimal('0.2')    ←データができています
```

この例から、特別な Decimal 型の数値となっていることがわかる。ただし、print 関数で出力すると一般的な形式の数値として確認できる。(次の例)

例. Decimal 型の数値の整形出力 (先の例の続き)

```
>>> print(d1)   
0.1    ← 一般的な書式  
>>> print(d2)   
0.2    ← 一般的な書式
```

これら d1, d2 の加算の例を次に示す。

例. Decimal 型数値の加算 (先の例の続き)

```
>>> d3 = d1 + d2     ←加算の実行  
>>> print(d3)     ←整形出力  
0.3    ←計算誤差が見られない
```

このように、丸めの処理を施していないにもかかわらず、正確な計算結果が得られていることがわかる。

正確な 10 進演算の必要性.

例えば、金融機関用のシステム開発においては、扱うデータ (各種金額の情報など) は 10 進数で表現され、その計算には誤差が許されない。従って、そのような分野では、decimal モジュールが提供するような正確な 10 進演算が必要とされる。

▲注意▲

先の例に示したように、Decimal コンストラクタに値を文字列で与えていることが重要である。この段階で値を float のリテラルとして与えると、その部分が float 型となり、それが原因で、Decimal の値にも誤差が混入してしまう。

例. 良くない例 (先の例の続き)

```
>>> Decimal(0.1)     ←数値リテラル (float) を与えると  
Decimal('0.1000000000000000055511151231257827021181583404541015625')    ←誤差が混入する
```

▲注意▲

Decimal 型の数値は、int 型の数値との演算はできるが、その他の型の数値 (float や complex, Fraction など) との演算はできないので注意すること。

```
>>> Decimal('2') + 3      Enter      ← int 型との演算は
Decimal('5')                ←可能

>>> Decimal('2') + 3.0    Enter      ← float 型との演算は
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    Decimal('2') + 3.0
    ~~~~~
TypeError: unsupported operand type(s) for +:  'decimal.Decimal' and 'float'
```

重要)

■ decimal モジュールの演算環境

数値の有効桁数.

例. 有効桁数の確認 (先の例の続き)

デフォルトの桁数は 28 桁に設定されている。この条件で除算を実行する例を次に示す。

```
>>> print( Decimal('10') / Decimal('3') )
```

Enter

3.3333333333333333333333333333 ← 28桁

prec 属性の値を変更することで、有効桁数を変えることができる。(次の例)

例. 有効桁数を 70 桁にする (先の例の続き)

[illegible]

有効桁数を超える演算の場合，結果は丸められ，指数表現となる．(次の例)

例. 有効桁数と演算結果 (先の例の続き)

```
>>> getcontext().prec = 5 Enter ← 5桁に変更
>>> Decimal('111111111') ** 2 Enter ←有効桁数を超える演算は
Decimal('1.2346E+16') ←丸めた上、指数表現となる
>>> getcontext().prec = 28 Enter ←デフォルトに戻す
>>> Decimal('111111111') ** 2 Enter ←上と同じ計算
Decimal('12345678987654321') ←正確な表現
```

数値の丸め.

演算環境の rounding 属性には丸めの手法（表 5）を設定する。

参考) 表5の ROUND 05UP は、チェックデジットの値を丸める際などで採用される丸め手法である。

32

表 5: 丸めの手法を指定する定数

定 数	解 説	定 数	解 説
ROUND_HALF_EVEN	偶数丸め	ROUND_HALF_UP	四捨五入
ROUND_HALF_DOWN	五捨六入	ROUND_UP	切り上げ
ROUND_DOWN	切り捨て	ROUND_CEILING	正の無限大方向に丸め
ROUND_FLOOR	負の無限大方向に丸め	ROUND_05UP (注)	0, 5は切り上げ, それ以外は切り捨て

(注) 切り上げの際は絶対値が大きくなる方向へ, 切り捨ての際は絶対値が小さくなる方向への丸め

例. 丸めの設定の確認 (先の例の続き)

```
>>> getcontext().rounding  Enter    ← 70 桁に変更
'ROUND_HALF_EVEN'          ← 偶数丸め (デフォルト)
```

Decimal の値に対して quantize メソッドを用いることで, 指定した桁数で丸めることができる。

書き方: Decimal オブジェクト.quantize(丸め位置のための参照値)

「丸め位置のための参照値」には Decimal オブジェクトなどの数値を与える。丸め位置は「丸め位置のための参照値」の指数部分に倣う。

quantize メソッドを使用して, 小数点以下の桁を丸める例を示す。

例. 小数点以下の桁の丸め (先の例の続き)

```
>>> x = Decimal('0.12345')  Enter    ← この値を
>>> x.quantize( Decimal('0.1') )  Enter    ← 小数点以下 1 桁に丸める
Decimal('0.1')                ← 結果
>>> x.quantize( Decimal('0.01') )  Enter    ← 小数点以下 2 桁に丸める
Decimal('0.12')               ← 結果
>>> x.quantize( Decimal('0.001') )  Enter    ← 小数点以下 3 桁に丸める
Decimal('0.123')              ← 結果
```

quantize メソッドを使用して, 整数部の桁を丸める例を示す。

例. 整数部の桁の丸め (先の例の続き)

```
>>> x = Decimal('54321')  Enter    ← この値を
>>> x.quantize( Decimal('1E+1') )  Enter    ← 10 の位に丸める
Decimal('5.432E+4')          ← 結果
>>> x.quantize( Decimal('1E+2') )  Enter    ← 100 の位に丸める
Decimal('5.43E+4')           ← 結果
>>> x.quantize( Decimal('1E+3') )  Enter    ← 1000 の位に丸める
Decimal('5.4E+4')            ← 結果
```

上の例の丸め処理の際に getcontext().rounding の設定が採用されている。quantize メソッドの 'rounding=' 引数に丸め手法を指定することもできる。

例. 手法を指定した丸め (先の例の続き)

```
>>> from decimal import ROUND_UP  Enter    ← 必要な丸めを読み込み
>>> x.quantize( Decimal('0.001'), rounding=ROUND_UP )  Enter    ← 小数点以下 3 桁に丸める (切り上げ)
Decimal('0.124')              ← 結果
```

値が持つ実際の小数点以下の桁より長い桁で丸めた場合は, 結果の値の桁が丸め指定の桁数まで伸びる。(次の例)

例. より長い桁数で丸める (先の例の続き)

```
>>> x = Decimal('0.12')  Enter    ← 小数点以下が 2 桁の場合に
>>> x.quantize( Decimal('0.0001') )  Enter    ← より長い桁数で丸めると
Decimal('0.1200')          ← 結果の桁数が伸びる
```

▲注意▲

上の例のようなケースで, 結果の値が有効桁数を超える場合はエラー (InvalidOperation) となる。(次の例)

例. 丸めの結果が有効桁数を超える場合（先の例の続き）

```
>>> getcontext().prec = 2  ←有効桁数を小さく（2 桁）設定して
>>> x = Decimal('0.12')  ←先と同じ
>>> x.quantize( Decimal('0.0001') )  ←丸め処理を行うと
Traceback (most recent call last):                               ←エラーとなる
  File "<stdin>", line 1, in <module>
    x.quantize( Decimal('0.0001') )
    ~~~~~
decimal.InvalidOperation: [
```

quantize メソッドの引数に関して詳しくは公式インターネットサイトを参照のこと。また、一時的な演算環境として localcontext も使えるが、これに関しても公式インターネットサイトを参照のこと。

■ Decimal 型の数値に対する数学関数

Decimal オブジェクトのメソッドの形でいくつかの数学関数（表 6）が使える。

表 6: Decimal オブジェクトに対する数学関数のメソッド（一部）

メソッド	解 説	メソッド	解 説
x.sqrt()	\sqrt{x}	x.exp()	e^x
x.ln()	$\log_e x$	x.log10()	$\log_{10} x$
x.max(y)	x, y の大きい方	x.min(y)	x, y の小さい方
x.to_integral_value()		x を適切に丸めて整数化	

x, y は Decimal オブジェクト

例. 数学関数

```
>>> from decimal import Decimal 
>>> x = Decimal('2') 
>>> x.sqrt()  ←  $\sqrt{2}$ 
Decimal('1.414213562373095048801688724')
>>> x = Decimal('1') 
>>> y = x.exp()  ← e
>>> y 
Decimal('2.718281828459045235360287471')
>>> y.ln()  ←  $\log_e e$ 
Decimal('0.9999999999999999999999999999')
```

```
>>> x = Decimal('1000') 
>>> x.log10()  ←  $\log_{10} 1000$ 
Decimal('3')
>>> x = Decimal('2')  ←小
>>> y = Decimal('3')  ←大
>>> x.max(y)  ←大きい方は
Decimal('3')
>>> x.min(y)  ←小さい方は
Decimal('2')
```

■ Decimal から int, float に変換する方法

Decimal に対して to_integral_value メソッドを使用することで、現在の演算環境の丸めモードに従って小数部を適切に処理した整数値（ただし型は Decimal）のオブジェクトが得られる。この結果に対して int コンストラクタを適用することで、int 型整数に変換できる。

小数点付きの Decimal 型オブジェクトは float コンストラクタで float 型の数値に変換することができる。ただし、float 型の有効桁数（およそ 15～17 桁程度）を超える場合は丸め誤差や精度の損失が発生するので、事前に Decimal の値を適切に丸めてから変換することが望ましい。

2.4.6.17 乱数の生成

乱数を生成するには random モジュールを使用する方法がある。このモジュールを使用するには次のようにする。

```
import random
```

■ 整数の乱数

整数の乱数を生成するには randrange メソッドを使用する。

書き方： `randrange(開始, 上限, 増分)`

「開始」以上「上限」未満の範囲で整数の乱数を生成する。生成される乱数に偏りは無く一様である。「増分」は省略可能であるが、これを与えると

開始 + $n \times$ 増分 (n は非負の整数)

の形で乱数を生成する。

randrange に引数を 1 つだけ与えると、上限のみを指定した動作となる。(0 以上「上限」未満の乱数)

例. 整数の乱数の生成

```
>>> import random [Enter] ←モジュールの読み込み
>>> random.randrange(0,100) [Enter] ← 0 以上 100 未満の整数の乱数の生成
16 ←得られた乱数
```

複数の乱数を生成する例を次に示す。

例. 複数の乱数の生成

```
>>> [ random.randrange(0,100) for i in range(10) ] [Enter] ← 0 以上 100 未満の整数乱数を 10 個生成
[16, 51, 90, 68, 82, 39, 83, 23, 15, 35] ←得られた乱数列
>>> [ random.randrange(0,100,10) for i in range(10) ] [Enter] ←増分を指定した乱数生成
[50, 90, 30, 80, 20, 40, 20, 10, 90, 40] ←得られた乱数列
>>> [ random.randrange(10) for i in range(10) ] [Enter] ←上限のみを指定した乱数生成
[5, 1, 3, 9, 3, 3, 2, 8, 7, 1] ←得られた乱数列
```

この例では、データ列が [~] で括られたリストという形で得られている。リストに関しては「2.5.1 リスト」(p.57) のところで説明する。また、この例で示した手法はリストの要素の内包表記を応用したものであり、詳しくは「2.6.1.7 for を使ったデータ構造の生成 (要素の内包表記)」(p.92) のところで説明する。

randrange とは別に、一様な整数の乱数を生成する randint もある。

書き方： `randint(最小値, 最大値)`

「最小値」以上「最大値」以下の範囲の乱数を生成する。

例. randint による整数乱数の生成

```
>>> [ random.randint(10,15) for i in range(10) ] [Enter] ← 10 以上 15 以下の整数乱数を 10 個生成
[15, 14, 11, 12, 10, 15, 10, 15, 12, 13] ←得られた乱数列
```

■ 浮動小数点数の乱数

浮動小数点数 (float) の一様乱数 (0 以上 1 未満) を生成するには random メソッドを使用する。

例. float の乱数の生成

```
>>> random.random() [Enter] ← float 乱数 (0 以上 1.0 未満) の生成
0.6830640714706756 ←得られた乱数
```

範囲を指定して一様な浮動小数点数の乱数を生成するには uniform を使用する。

書き方： `uniform(開始, 上限)`

「開始」以上「上限」未満の範囲の乱数を生成する。

例. uniform 関数による乱数の生成

```
>>> [ random.uniform(10,15) for i in range(10) ] [Enter] ← float 乱数列 (10 以上 15 未満) の生成
[13.518212730827601, 10.314921371305998, 14.58509484557475, 11.108519481070932,
14.016725270731012, 10.712471969544122, 12.714949788450062, 10.45607976915113,
14.966107767822391, 14.375436436680728] ←↑ 得られた乱数列
```


■ 乱数生成における状態、種に関すること

randrange, randint, random, uniform はメルセンヌ・ツイスタ⁴⁷ を用いたものであり、乱数生成が確定的である。すなわち、ある初期状態から生成する乱数の並びが決められている。このような乱数を疑似乱数⁴⁸ という。疑似乱数の生成が確定的であることは次の例で確かめることができる。

例. 同じ初期状態から同じ乱数列が生成される例

```
>>> random.seed(0)  [Enter]  ←乱数生成処理を初期化
>>> [ random.randrange(0,100) for i in range(10) ]  [Enter]  ←乱数を 10 個生成
[49, 97, 53, 5, 33, 65, 62, 51, 38, 61]  ←得られた乱数
>>> random.seed(0)  [Enter]  ←乱数生成処理を再度初期化
>>> [ random.randrange(0,100) for i in range(10) ]  [Enter]  ←乱数を 10 個生成
[49, 97, 53, 5, 33, 65, 62, 51, 38, 61]  ←同じ乱数列が得られている
```

この例では seed 関数を使用して乱数生成の状態を初期化している。この関数の引数には、乱数の初期状態を意味する種 (seed) を与える。同一の種で初期化すると、同一の乱数生成過程となる。また、異なる種からは異なる乱数の系列が生成される。

このように、指定した初期状態から確定的な乱数を生成することは、統計学や機械学習のためのプログラム開発などにおいて、決まったテストデータを生成する場合に必要となる。また、次のような関数を使用すると、疑似乱数生成におけるある時点での状態を取得し、その状態を再現することができる。

状態の取得: `getstate()`
状態の再現: `setstate(状態)`

getstate は実行時点の乱数生成の状態を意味するオブジェクトを返す。このオブジェクトを setstate の引数に与えて実行すると、乱数生成の系列を再現することができる。(次の例)

例. 乱数生成の状態の取得

```
>>> [ random.randrange(10) for i in range(10) ]  [Enter]  ←乱数生成
[3, 9, 8, 2, 5, 9, 7, 9, 1, 9]  ←得られた乱数列
>>> st = random.getstate()  [Enter]  ←この時点の状態を st に取得
>>> [ random.randrange(10) for i in range(10) ]  [Enter]  ←後続の乱数生成
[0, 7, 4, 8, 3, 3, 7, 8, 8, 7]  ←得られた乱数列
```

この例で取得した st を使用して、再度その時点での乱数生成を再現することができる。(次の例)

例. 乱数生成の状態の再現 (先の例の続き)

```
>>> random.setstate(st)  [Enter]  ← st が示す状態を再現
>>> [ random.randrange(10) for i in range(10) ]  [Enter]  ←乱数生成
[0, 7, 4, 8, 3, 3, 7, 8, 8, 7]  ←再現されている
```

■ 正規乱数

gauss 関数を使用すると正規分布 (ガウス分布) に沿った乱数を生成することができる。

書き方: `gauss(μ , σ)`

「 μ 」を平均, 「 σ 」を標準偏差とする正規分布 $N[\mu, \sigma]$ に沿った乱数を返す。

例. $N[0, 1]$ に沿った正規乱数を 100,000 個生成して度数分布を調べる

```
>>> nr = [ random.gauss(0,1) for i in range(100000) ]  [Enter]  ←正規乱数 100,000 個生成
>>> import matplotlib.pyplot as plt  [Enter]  ←可視化ライブラリの読み込み
>>> h = plt.hist(nr, bins=16)  [Enter]  ←ヒストグラムの作成
>>> plt.show()  [Enter]  ←作図の実行
```

これは、可視化ライブラリ matplotlib⁴⁹ を用いて乱数のリストのヒストグラムを作成する例であり、この処理の直後

⁴⁷ 乱数生成器の 1 つ。参考文献: M. Matsumoto, T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol.8, No.1, January pp.3-30 (1998)

⁴⁸ 生成系列が確定的でない (再現性がない) 乱数を真の乱数, 真性乱数と呼ぶ。

⁴⁹ PSF 標準のライブラリではないので、計算機環境に別途インストールしておく必要がある。

に図 4 の (a) のようなヒストグラムが表示される。

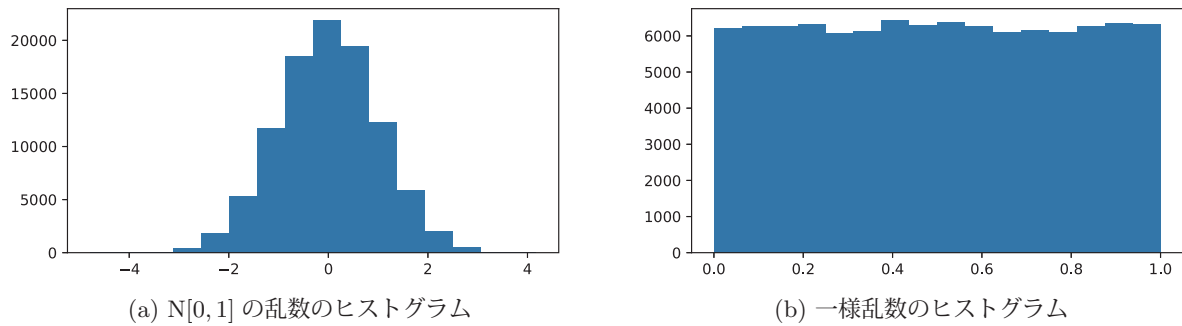


図 4: 100,000 個の乱数のヒストグラム

参考.

正規分布は μ の近くにサンプルが集中するものであり、先の一様乱数とは異なる。参考までに一様乱数のヒストグラムを作成する処理を次に示す。

例. 一様乱数を 100,000 個生成して度数分布を調べる (先の例の続き)

```
>>> ur = [ random.random() for i in range(100000) ] Enter ←一様乱数 100,000 個生成
>>> h = plt.hist(ur, bins=16) Enter ←ヒストグラムの作成
>>> plt.show() Enter ←作図の実行
```

この処理の結果、図 4 の (b) のようなヒストグラムが表示される。

※ matplotlib に関することは別の文献⁵⁰ を参照されたい。

random モジュールは本書で紹介したもの以外にも乱数を生成する関数を提供している。詳しくは公式ドキュメントを参照のこと。

■ 安全な乱数

random モジュールで生成する乱数は、種から予測できるものである。従って、暗号的な処理においては random モジュールが提供する乱数生成機能を使用してはならない。より安全な乱数生成には secrets モジュールを使用すべきである。このモジュールは、暗号関連処理のために OS が提供する機能⁵¹ を利用して乱数を生成する。

secrets モジュールを使用するためには次のようにする。

```
import secrets
```

ここでは、secrets モジュールが提供する乱数発生機能を紹介する。

1) randbelow(n)

0 以上 n 未満の整数の乱数 (一様乱数) を生成する。 (n は int 型で与える)

例. 整数の乱数の発生

```
>>> secrets.randbelow(100) Enter ← 0 以上 100 未満の整数の乱数の発生
52 ←得られた乱数
```

2) randbits(n)

0 以上 $2^n - 1$ 以下の整数の乱数を一様に生成する。 (n は int 型で与える)

例. ビット長指定の乱数の発生

```
>>> secrets.randbits(16) Enter ← 0 以上  $2^{16} - 1$  (=65,535) 以下の整数の乱数の発生
3360 ←得られた乱数
```

ここで示したものの以外にも secrets モジュールは便利な機能を提供している。詳しくは Python の公式インターネットサイトなどを参照のこと。

⁵⁰ 拙書「Python3 ライブラリブック」でも解説しています。

⁵¹ 「暗号的に安全な擬似乱数生成器」 (CSPRNG : cryptographically secure pseudo random number generator)

2.4.7 文字列

文字列はダブルクォート「"」もしくはシングルクォート「'」といった引用符で括ったデータである。どちらを使っても良い⁵² が、引用符の開始と終了は同じものにしなければならない。

文字列の例. 'Python', "パイソン"

またこのように、引用符で括って文字列データを表現したものを**文字列リテラル**という。

2.4.7.1 エスケープシーケンス

Python では C 言語での扱いと同様に、文字列中には特殊な機能を持った記号（エスケープシーケンス）を含めることができる。エスケープ記号は「¥」（半角）である。これは、処理環境によってはバックスラッシュ「\」の表示になる場合⁵³ がある。

《 ¥ と \ に関する注意事項》

Unicode の 165 (0xa5) に対応する文字も「¥」と表示される場合がある。この場合の「¥」はエスケープシーケンスの「\」とは異なるので注意が必要である。特に Apple 社の Macintosh の機種によってはエスケープシーケンスの「\」を入力する際に Option + ¥ と入力する場合がある。

一般的な意味では「文字列」は可読な記号列のことを指すが、記号列の中に特別な働きを持った、ある種の「制御記号」を含めると、その文字列を端末画面などに表示する際の制御ができる。例えば、「¥n」というエスケープシーケンスを含んだ文字列

”一行目の内容です。 ¥n 二行目の内容です。”
を端末画面に表示してみる。

```
>>> a = "一行目の内容です。 ¥n 二行目の内容です。" Enter
>>> print( a ) Enter
一行目の内容です。
二行目の内容です。
```

このように、「¥n」の位置で文字列の表示が改行されていることがわかる。この例における print は端末装置に出力する（画面に表示する）ための機能であるが、詳しくは以後の章で解説する。

注) print 関数を使用せずに直接的に文字列を対話モードで参照した場合はエスケープシーケンスの効果は表れない。

例. 文字列の内容を直接参照する（先の例の続き）

```
>>> a Enter ←変数 a の内容を参照
'一行目の内容です。 ¥n 二行目の内容です。'
```

このように、文字列の内容がそのまま表示される。

参考) テキストファイルの中の改行コードに関しては「2.7.3 ファイルからの入力」(p.108)で解説する。

表示に使用する端末装置によっては、ベルを鳴らすことも可能である。例えば次のように「¥a」というエスケープシーケンスを含む文字列を print する。

例. 「¥a」でベル音を鳴らす

```
>>> a = "Ring the bell.¥a" Enter
>>> print( a ) Enter
Ring the bell.
```

この表示と同時にベル音が鳴る。(端末装置によって音は違う。あるいは音が鳴らない場合もある)

エスケープシーケンスには様々なものがあり、代表的なものを表 7 に挙げる。

この他にも Python では u でエスケープするシーケンスも使用できる。「¥u」に続いて Unicode の 16 進数表記を記述すると、その文字コードに該当する文字となる。

⁵²Python 処理系はシングルクォートを付けて文字列を表示する。

⁵³「¥」、「\」と表示が異なっているが ASCII コードとしては共に 92 (0x5c) である。

表 7: 代表的なエスケープシーケンス

ESC	機能	ESC	機能
¥n	改行	¥t	タブ
¥r	行頭にカーソルを復帰	¥v	垂直タブ
¥f	フォームフィード	¥b	バックスペース
¥a	ベル	¥¥	'¥' そのもの
¥"	ダブルクオート文字	¥'	シングルクオート文字

例. エスケープシーケンスによる Unicode 文字の表現

```
>>> '¥u604b¥u611b'  [Enter]    ← Unicode の文字コードで文字列を表現
'恋愛'                ← 結果としての文字列
```

参考) ANSI エスケープシーケンス (ANSI コード)

コンソール (ターミナルウィンドウ) によっては **ANSI エスケープシーケンス** (ANSI コード) を使うことができる。 (ウィンドウ環境によっては有効にならないことがあるので注意)

例. 下線を付ける

```
>>> a = '¥033[4m 下線¥033[0m'  [Enter]    ← '¥033[4m ~ ¥033[0m' で下線を付ける
>>> print( a )  [Enter]
下線
```

¥033[4m 以降の文字に下線が付く。 ¥033[0m で ANSI エスケープシーケンスによる装飾を解除する。

例. 色を付ける

```
>>> a = '¥033[31m 赤い文字¥033[0m'  [Enter]    ← '¥033[31m ~ ¥033[0m' で赤にする
>>> print( a )  [Enter]
赤い文字
```

¥033[31m 以降の文字が赤くなる。 ¥033[0m で ANSI エスケープシーケンスによる装飾を解除する。

拡張された絵文字などは「¥u」ではなく、「¥U」でエスケープして 8 桁の 16 進数で表現する。

例. 顔文字 😊

```
>>> a = '¥U0001f600'  [Enter]    ← '😊' の Unicode のエスケープ表現
>>> print(a)  [Enter]    ← 出力する
😊                ← 結果
```

拡張された文字に関しては、後の「Unicode の文字符号化モデル」(p.47) で説明する。

この他にも多くの ANSI エスケープシーケンスが存在するが、詳細については他の資料を参照のこと。

2.4.7.2 raw 文字列 (raw string)

文字列中のエスケープシーケンスを無視して '¥' を単なる文字として扱うには **raw 文字列** として扱うと良い。通常の文字列の先頭に 'r' もしくは 'R' を付けるとその文字列に含まれる記号は全て「そのままの記号」として扱われる。

例. raw 文字列

```
>>> s = r'abc¥ndef'  [Enter]    ← raw 文字列の生成
>>> print(s)  [Enter]    ← 出力してみる
abc¥ndef          ← '¥n' で改行されず、そのままの文字として扱われている
```

2.4.7.3 複数行に渡る文字列

単引用符 3 つ ''' ~ ''' で括ることで、複数の行に渡る文字列を記述することができる。

例. 複数行に渡る文字列の記述

```
>>> s = ''' これは  ←記述の開始
...   複数の行にわたる 
...   文字列です. '''  ←記述の終了
```

これで3行に渡る文字列がsに得られた。次にこの内容を確認する。

例. 複数行に渡る文字列の確認（先の例の続き）

```
>>> s  ← s の内容確認
' これは¥n 複数の行にわたる ¥n 文字列です. ' ← s の内容表示

>>> print( s )  ← s の整形表示
これは
複数の行にわたる      ←複数の行として表示されている
文字列です.
```

2.4.7.4 文字列の長さ（文字数）

文字列の長さ（文字列を構成する文字の個数）はlen関数で求めることができる。

例. 文字列の長さを求める

```
>>> s = 'abcdefg'  ← 7文字で構成される文字列
>>> len( s )  ← len関数の実行
7 ← 長さ（文字数）が得られた
```

2.4.7.5 文字列リテラルの接続

文字列リテラル（引用符で括った文字列の表現）を接続すると1つの文字列となる。

例. 文字列リテラルの接続（その1）

```
>>> 'abc' 'def'  ← 接続 (1)
'abcdef' ← 結果
>>> 'abc''def'  ← 接続 (2)
'abcdef' ← 結果（先と同じ）
```

例. 文字列リテラルの接続（その2）

```
>>> '''abc''' '''def'''  ← 接続 (3)
'abcdef' ← 結果
>>> '''abc''''''def'''  ← 接続 (4)
'abcdef' ← 結果（先と同じ）
```

一見して視認しにくい場合があるので注意すること。

2.4.7.6 文字列の分解と合成

文字列の指定した部分を取り出すには「[]」で括った添字（スライス）を付ける。例えば、'abcdef' という文字列があった場合、'abcdef'[2] は 'c' である。このように位置を意味する添字を付けることで、文字列の部分を取り出すことができる。先頭の文字の添字は0である。（インデックスの開始は「0」）

《 部分文字列の取り出し 》

書き方 (1): 文字列 [n]

「文字列」のn番目の1文字を取り出す。

書き方 (2): 文字列 [n₁:n₂]

「文字列」のn₁番目からn₂ - 1番目までの部分を取り出す。

書き方 (3): 文字列 [n₁:n₂:n₃]

上記のことに加え、インデックスの増分n₃を指定して要素を取り出す。

※ 添字（スライス）の更に高度な応用については「2.5.5 添字（スライス）の高度な応用」（p.83）を参照のこと。

例. 1 文字の取り出し

```
>>> s = '美味しい食べ物' Enter ←文字列の生成
>>> s[1] Enter ←インデックス 1 番目の文字の取得
'味' ←取り出した部分文字列
```

例. 部分文字列の取得

```
>>> s = 'abcdefg'  Enter    ←文字列の生成
>>> s[2:5]         Enter    ← 2 番目から 5-1 番目までの部分文字列の取得
'cde'              ←取り出した部分文字列
```

例. インデックスの増分を指定して要素を取り出す

```
>>> s = '0123456' Enter ←文字列の生成
>>> s[1:6:2] Enter ←1 番目から 6-1 番目までの要素を 1 つ飛ばしながら抽出
'135' ←取り出した要素から成る文字列
```

文字列は事後にその内容を変更することができない⁵⁴。

例. 文字列を事後に変更する試み

```
>>> s = 'ABCdEF'      Enter      ←文字列の生成
>>> s[3] = 'D'        Enter      ←それを変更しようとする…
Traceback (most recent call last):  ←エラーとなる
  File "<stdin>", line 1, in <module>
    s[3] = 'D'
    ~~~~
TypeError: 'str' object does not support item assignment
```

従って、文字列を変更する場合は新規に作成する必要がある。(次の例)

例. 文字列の内容の新規作成 (先の例の続き)

```
>>> s = 'ABCDEF'      Enter      ←文字列の新規作成
>>> s                  Enter      ←内容確認
'ABCDEF'
```

■ 文字列の連結と繰り返し

文字列の連結には、`'+'` による加算表現が使える。

例. 文字列の連結 (1)

```
>>> 'abc' + 'def'      Enter      ←加算記号で連結
'abcdef'                ←連結結果
```

更に、累算代入 '+=' を使って次々と累積的に連結してゆく⁵⁵ こともできる。

例. 文字列の連結 (2)

```
>>> s = 'abc'      Enter      ←文字列の生成
>>> s += 'def'     Enter      ←'def'を追加
>>> s += 'ghi'     Enter      ←'ghi'を追加
>>> s              Enter      ←内容確認
'abcdefghijklmno'   ←最終的な内容
```

積の演算 '*' を使うと文字列の繰り返しを生成できる。

例. 文字列の繰り返し

```
>>> 'abc' * 10      Enter      ← 'abc' を 10 回繰り返す
'abcabcabcabcabcabcabcabcabcabc' ← 生成結果
```

■ split メソッドによる文字列の分解

split メソッドを使用すると、特定の文字（列）を区切りとして文字列を分解することができる。

⁵⁴ このことをイミュータブルであるという。「イミュータブル」については後の「ミュータブル／イミュータブル」(p.69)で解説する。

⁵⁵ 若干の注意点がある。詳しくは後の「2.5.11 累算代入によるデータ構造の拡張に関する注意点」(p.88)を参照のこと。

例. 文字列の分解

```
>>> s = 'ab,cd,ef,gh' Enter ←文字列の生成
>>> s.split(',') Enter ←コンマ','を境(区切り)にして文字列を分解
['ab', 'cd', 'ef', 'gh'] ←分解されたリスト
```

分解されたものがリストの形で得られている。リストに関しては「2.5.1 リスト」(p.57) のところで説明する。

参考) 分解の回数を指定する方法

split メソッドで文字列を分解する際、キーワード引数 'maxsplit=' に分解の回数を指定することができる。

例. 2 回だけ分解する (先の例の続き)

```
>>> s.split(',', maxsplit=2) Enter ←回数を設定して分解
['ab', 'cd', 'ef,gh'] ←分解結果
```

この例から、分解されない要素が残ることがわかる。

split メソッドに引数を与えない場合は、対象の文字列中の空白文字を区切りとして分解する。

例. 引数なしで split を実行

```
>>> s = 'abc def ghi\tjkl\tnmno' Enter ←各種の空白文字を含む文字列
>>> print(s) Enter ←内容確認
abc def ghi jkl ←結果表示
mno
>>> s.split() Enter ←引数なしで split を s に対して実行
['abc', 'def', 'ghi', 'jkl', 'mno'] ←結果表示
```

空白文字はタブや改行などのエスケープシーケンスを含む。

■ splitlines メソッドによる行の分離

splitlines メソッドを使用すると、複数行に渡る文字列の行を分離することができる。

例. 行の分離

```
>>> s = '''これは Enter ←3行にわたる文字列
... 複数の行にわたる Enter
... 文字列です。''' Enter
>>> a = s.splitlines() Enter ←行を分離
>>> a Enter ←内容確認
['これは', '複数の行にわたる', '文字列です。'] ←結果表示(リストになっている)
```

split メソッドを用いて '\n' を区切りにして分解することもできるが、行の区切りは '\n' 以外にもあるので、行の分離には splitlines を使用するべきである。

■ リストの要素の連結

join メソッドを使用すると、リストの要素を全て連結することができる。(split メソッドの逆の処理)

ただしその場合のリストの要素は全て文字列型でなければならない。

例. リストの要素の連結 (split の逆)

```
>>> lst = ['a', 'b', 'c', 'd'] Enter ←リストの生成
>>> ':'.join(lst) Enter ←コロン ':' を境(区切り)にして文字列を連結
'a:b:c:d' ←リストを連結した文字列
```

join メソッドは区切り文字となる文字列型データに対して実行し、その引数に連結対象要素をもつリストを与える。

2.4.7.7 文字列の置換

文字列中の特定の部分を置き換えるには replace メソッドを使用する。

《 文字列の置換 》

書き方： 文字列.replace(対象部分, 置換後の文字列)

「文字列」の中の「対象部分」を探し、それを「置換後の文字列」に置き換えたものを返す。元の文字列は変更されない。

例. 文字列の置換

```
>>> a = 'abcdefgabcdefgabcdefg' Enter ←元の文字列
>>> a.replace('bcd','BCD') Enter ← 'bcd' を 'BCD' に置き換える
'aBCDefgaBCDefgaBCDefg' ←置換結果
```

2.4.7.8 文字の置換

1 文字単位で置換する規則に基いて、文字列中の文字を置換することができる。手順としては、まず str クラスの maketrans メソッドを用いて置換規則のオブジェクトを生成し、それを用いて translate メソッドによって置換処理を行う。

例. 1 文字単位の変換規則の適用

```
>>> tr = str.maketrans('ABC','abc') Enter ←置換規則を tr として生成
>>> 'DAEBFCG'.translate(tr) Enter ← tr を用いて置換処理
'DaEbFcG' ←処理結果
```

この例における tr は、

A	B	C
↓	↓	↓
a	b	c

と置換する規則であり、その内容は辞書オブジェクト (p.76 「2.5.4 辞書型」で解説する) である。translate メソッドは置換結果の文字列を返す。

この方法では日本語を含む多バイト系文字の置換も可能である。特にアルファベット大文字／小文字の間での変換に関しては後の「2.4.7.10 大文字／小文字の変換と判定」(p.44) で説明する方法が便利である。

2.4.7.9 英字, 数字の判定

文字列に含まれる記号の種類を判定するいくつかの方法を表 8 に示す。

表 8: 文字列に含まれる文字種を判定するメソッド (一部)

メソッド	説明
isalpha()	対象の文字列の要素が全てアルファベット
isdecimal()	対象の文字列の要素が全て数字
isalnum()	対象の文字列の要素が全てアルファベットか数字

これらメソッドの実行例を示す。

例. 文字列の要素が全てアルファベットかどうかの判定

```
>>> 'abc'.isalpha() Enter
True
>>> 'aBc'.isalpha() Enter
True
>>> 'm4a'.isalpha() Enter
False
```

例. 要素が全て数字かどうかの判定

```
>>> '1234'.isdecimal() Enter
True
>>> '640x480'.isdecimal() Enter
False
```


例. 要素が全てアルファベットか数字かの判定

```
>>> '123'.isalnum() Enter
True
>>> '640x480'.isalnum() Enter
True
>>> '123+456'.isalnum() Enter
False
```

2.4.7.10 大文字／小文字の変換と判定

アルファベット大文字／小文字の変換や判定のためのメソッド群が使用できる。

例. 文字列中の小文字を大文字に変換する処理

```
>>> s = 'this is a sample.' Enter ←小文字による文字列
>>> t = s.upper() Enter ←小文字を大文字に変換する処理
>>> t Enter ←内容の確認
'THIS IS A SAMPLE.' ←大文字になっている
```

この例では upper メソッドを用いて小文字を大文字に変換している。このメソッドは変換元の文字列オブジェクトに対して適用し、変換結果の文字列を返す。また、元の文字列は変更されない。

他にも様々なメソッドがある。表 9 にそれらの一部を示す。

表 9: 大文字／小文字の変換や判定

メソッド	元の文字列	処理結果 (戻り値)	説明
upper()	'this is a sample.'	'THIS IS A SAMPLE.'	小文字→大文字
lower()	'THIS IS A SAMPLE.'	'this is a sample.'	大文字→小文字
capitalize()	'this is a sample.'	'This is a sample.'	文頭を大文字に変換
title()	'this is a sample.'	'This Is A Sample.'	各単語の先頭を大文字に変換
swapcase()	'This Is A Sample.'	'tHIS iS a sAMPLE.'	小文字/大文字を逆転
islower()	'this is a sample.'	True	全て小文字の場合に真 小文字以外を含むと偽
	'This Is A Sample.'	False	
isupper()	'THIS IS A SAMPLE.'	True	全て大文字の場合に真 大文字以外を含むと偽
	'This Is A Sample.'	False	

元の文字列. メソッド の形で実行する。

2.4.7.11 文字列の含有検査

文字列の中に「ある文字列」が含まれるかどうかを検査するには in 演算子を使う。

書き方： 探したい文字列 in 元の文字列

このようにすることで「元の文字列」の中に「探したい文字列」があるかどうかを判定できる。

例. 文字列の含有検査

```
>>> s = '私は大阪府に住んでいます。' Enter ←元の文字列の生成
>>> '大阪' in s Enter ←「大阪」が含まれるか検査
True ←含まれる
>>> '東京' in s Enter ←「東京」が含まれるか検査
False ←含まれない
```

結果は「真か偽か」を表現する**真理値**であり、True（真）か False（偽）の値である。真理値に関しては「2.4.8 真理値」(p.50) で解説する。

in を用いた文字列の含有検査を応用すると、様々な文字列編集が可能となる。

応用例. 冗長な空白文字の除去

```
>>> s = 'A      B      C      D'  Enter    ←冗長な空白を含んだ文字列
>>> while ' ' in s: Enter              ←2つの空白文字' 'が存在する間の繰り返し
...     s = s.replace(' ', ' ') Enter  ←2つの空白文字' 'を1つの空白文字に置き換える
...     Enter    ←繰り返し処理の終了
>>> s Enter      ←内容確認
'A B C D'        ←空白文字が短縮されている
```

これは、重複する2つの空白文字を1つの空白文字に変換する処理を繰り返す例⁵⁶である。(while文による繰り返しに関してはp.89「2.6 制御構造」で解説する)

2.4.7.12 文字列の検索

文字列の中に「ある文字列」がどの位置にあるかを調べるには find メソッドを使用する。

書き方 (1): 文字列.find(語)

書き方 (2): 文字列.find(語, 開始位置, 終了位置)

「文字列」の中の「語」がある位置 (インデックス) を返す。検索は「文字列」の前方 (インデックスの小さい方) から後方 (インデックスの大きい方) に向かって行う。「文字列」の中での検索範囲は「開始位置」～「終了位置-1」で指定することができる。「終了位置」の記述を省略すると「開始位置」から「文字列」の末尾までが検索対象となる。「語」が見つからなければ -1 を返す。

例. find メソッドによる検索

```
>>> s = '0123word89word456word123' Enter    ←文字列 s を作成
>>> s.find('word') Enter                    ← s の中の 'word' の位置を求める
4                                           ←検出位置 (インデックス)
>>> s.find('test') Enter                    ← s の中の 'test' の位置を求める試み
-1                                           ←検出せず
```

例. 範囲を指定した検索 (先の例の続き)

```
>>> s.find('word', 8, 15) Enter              ← s の中のインデックス範囲 8~14で検索
10                                           ←検出位置 (インデックス)
>>> s.find('word', 8, 13) Enter              ← s の中のインデックス範囲 8~12で検索
-1                                           ←検出せず
```

「文字列」の後方から前方に向かって「語」を検索するメソッド rfind も使用できる。引数の与え方は find と同様である。

例. rfind メソッドによる検索 (先の例の続き)

```
>>> s.rfind('word') Enter                   ← s の末尾から先頭にかけて 'word' の位置を探す
17                                           ←検出位置 (インデックス)
>>> s.rfind('word', 0, 15) Enter            ← インデックス位置 14 から先頭にかけて 'word' の位置を探す
10                                           ←検出位置 (インデックス)
```

文字列検索には index メソッドも使用できる。

例. index メソッドによる検索 (先の例の続き)

```
>>> s.index('word') Enter                   ←検索が成功する場合
4                                           ←検出位置のインデックスが得られる
>>> s.index('test') Enter                   ←検索が失敗する場合
Traceback (most recent call last):         ←エラーとなる
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

この例からもわかるように、index メソッドで検索が失敗するとエラー (ValueError) が発生する。エラーが発生する事象を正しく扱うには後の「2.5.1.6 例外処理」(p.63) で説明する例外処理を施すべきである。

⁵⁶このアルゴリズムは最適ではない。更に高速なアルゴリズムについて考察されたい。

更に高度な文字列検査の方法に関しては「4.3 文字列検索と正規表現」(p.247)で解説する。

■ 文字列の左端／右端にあるものの検査

文字列の左端に指定した文字列部分があるかどうかを調べるには `startswith` メソッドを使用する。同様に右端にあるかどうかを調べるには `endswith` メソッドを使用する。

例. 左端／右端に指定した文字列があるかどうかを検査する

```
>>> s = 'ABCdefghiJKL'  Enter    ←検査対象文字列
>>> s.startswith('ABC')  Enter    ←左端に 'ABC' が
True                      ←ある
>>> s.endswith('ABC')    Enter    ←右端に 'ABC' は
False                     ←ない
>>> s.startswith('JKL')  Enter    ←左端に 'JKL' は
False                     ←ない
>>> s.endswith('JKL')    Enter    ←右端に 'JKL' が
True                      ←ある
```

■ 部分文字列の出現回数をカウントする

`count` メソッドを使用すると、文字列の中に現れる部分文字列の個数を数えることができる。

例. 部分文字列のカウント

```
>>> s = 'ABCdefABCghi'  Enter    ←この文字列の中に含まれる
>>> s.count('ABC')       Enter    ←'ABC' は
2                          ← 2 個
```

2.4.7.13 両端の文字の除去

文字列の両端にある特定の文字を除去するには `strip` メソッドを使用する。

書き方： `文字列.strip(除去対象の文字の並び)`

「文字列」の両端に「除去対象の文字の並び」に含まれる文字があればそれらを除去したものを返す。元の「文字列」は変化しない⁵⁷。

例. 両端にある特定の文字の除去

```
>>> s = 'ABCabcABCdefABC'  Enter    ←この文字列の両端にある
>>> s.strip('ABC')          Enter    ←'ABC' のどれかに該当する文字を除去する
'abcABCdef'                 ←処理結果
```

この例からわかるように両端に位置していないものは除去されない。

`strip` メソッドの引数に与える文字列は、除去対象の文字を並べたものであり、それら文字の順序は任意で良い。

例. 上と同様の例（先の例の続き）

```
>>> s.strip('CBA')          Enter    ←'CBA' のどれかに該当する文字を両端から除去する
'abcABCdef'                 ←処理結果
```

`strip` メソッドの引数を省略すると空白文字（スペース、タブ、改行文字など）を除去する。

例. 両端の空白文字を削除する

```
>>> s = '    abcdef¥t¥n'  Enter    ←この文字列の両端にある
>>> s.strip()              Enter    ←空白文字を除去する
'abcdef'                   ←処理結果
```

`strip` メソッド以外にも、左端のみを除去対象とする `lstrip`、右端のみを除去対象とする `rstrip` メソッドもある。また、「¥n」によって複数行として構成された文字列の整形に関して、後の「4.24 文字列の整形処理（分割、折り返しなど）： `textwrap` モジュール」(p.347)で解説する。

⁵⁷文字列型 (str) はイミュータブル（変更不可）なので当然であるとも言える。「イミュータブル」については後の「ミュータブル／イミュータブル」(p.69)で解説する。

2.4.7.14 文字コード、文字の種別に関すること

コンピュータで取り扱う文字は、記憶媒体上ではそれに対応するバイト値やバイト列として表現されている。例えば代表的な半角文字であるアスキー文字（ASCII）は巻末付録「J.2 アスキーコード表」（p.466）に示すように 0 ～ 127 の値としてコンピュータ内部では扱われている。例えば、アスキー文字「A」の文字コードは「65」であるという。

「文字コード」という言葉には下に示す 2 種類の意味があるので、この言葉がどちらを意味するかは、用いられる状況の文脈で判断すること。

- 1) ある文字に対応する数値（コードポイントの値⁵⁸）やバイト列
- 2) 文字と数値（あるいはバイト列）の対応関係を定める規則

特に上記 2) を意味する場合は、曖昧さを避けるために文字コード体系あるいはエンコーディングと呼ぶ場合がある。Python 言語で取り扱うことができるエンコーディングとしては、p.6 の表 1 に示したものをはじめ多くのものがあるが、Python 言語処理系内部では文字の取り扱いには Unicode⁵⁹ に基づいている。（Unicode は ASCII の体系を含んでいる）

ある 1 文字の文字コードを取得するには ord 関数を使用する。また逆に、文字コードを表す整数値から文字を取得するには chr 関数を使用する。

例. 文字コードの取得

```
>>> ord('a')  Enter  ← 'a'（半角文字）の文字コードの取得
97           ←得られた文字コード（ASCII コード）
>>> ord('あ')  Enter  ← 'あ'（全角文字）の文字コードの取得
12354        ←得られた文字コード（Unicode）
```

例. 文字コードに対応する文字の取得

```
>>> chr(97)  Enter  ←文字コード 97（10 進数）に対応する文字の取得
'a'         ←得られた文字
>>> chr(12354) Enter  ←文字コード 12354（10 進数）に対応する文字（Unicode）の取得
'あ'        ←得られた文字
```

■ 文字の種別の調査

標準モジュールの unicodedata を使用すると文字の種別を判定することができる。具体的にはこのモジュールの name 関数を使用して、指定した文字の**名前**（文字の属性）を取得する。

例. 文字の種別の判定

```
>>> import unicodedata  Enter  ←モジュールの読み込み
>>> unicodedata.name('a') Enter  ← 'a' の判定
'LATIN SMALL LETTER A'  ←判定結果：「半角英数小文字」
>>> unicodedata.name('A') Enter  ← 'A' の判定
'LATIN CAPITAL LETTER A' ←判定結果：「半角英数大文字」
>>> unicodedata.name('ア') Enter  ← 'ア' の判定
'HALFWIDTH KATAKANA LETTER A' ←判定結果：「半角カタカナ」
>>> unicodedata.name('あ') Enter  ← 'あ' の判定
'HIRAGANA LETTER A'        ←判定結果：「全角ひらがな」
>>> unicodedata.name('ア') Enter  ← 'ア' の判定
'KATAKANA LETTER A'        ←判定結果：「全角カタカナ」
>>> unicodedata.name('中') Enter  ← '中' の判定
'CJK UNIFIED IDEOGRAPH-4E2D' ←判定結果：「全角漢字」
```

name 関数の戻り値を構成する語の意味などについては Unicode の公式インターネットサイトを参照のこと。

■ Unicode の文字符号化モデル

Unicode は文字コード 0～1114111（16 進数で 0～10FFFF）の範囲で定義されており、特に Unicode においては文

⁵⁸後の「Unicode の文字符号化モデル」（p.47）で述べる。

⁵⁹詳しくは Unicode の公式インターネットサイト The Unicode Consortium（<http://www.unicode.org/>）で公開されている情報を参照のこと。

字に対する整数値（文字コード）のことを「コードポイントの値」と呼ぶ。本書執筆時点では、この範囲の全てのコードポイントに対しては Unicode 文字は割り当てられていない。例えば、コードポイントの最大値 1114111 の Unicode 文字を調べようとする次のようになる。

試み） コードポイント 1114111 の Unicode 文字について調べる

```
>>> c = chr(1114111) Enter ←コードポイント 111411 の文字を c に取得
>>> print( c ) Enter ←表示を試みる
.. ←不可解な表示（文字が割り当てられていない）
>>> unicodedata.name(c) Enter ← name 関数でこの文字の属性を調べてみると
Traceback (most recent call last): ←文字が定義されておらず
  File "<stdin>", line 1, in <module> エラーとなる
ValueError: no such name
```

name 関数の第 2 引数には例外時（文字が割り当てられていない場合）の戻り値を与えることができ、これによりエラー（例外）の発生を回避できる。

先の試みの続き）

```
>>> unicodedata.name(c, '未定義です') Enter ←再度 name 関数でこの文字の属性を調査
'未定義です' ←結果表示
```

参考） 使用できる全ての Unicode 文字を列挙するサンプルプログラムを「I.6 全ての Unicode 文字の列挙」（p.464）に示す。

Unicode のコードポイントの値の範囲は Plane0～Plane16 の 17 カテゴリーの平面（Plane）に分類されており、最も基本的な Plane0（コードポイント 0x0000～0xffff の平面）は**基本多言語面**（BMP：Basic Multilingual Plane）と呼ばれる。基本多言語面以外の平面はまとめて**補助平面**（Supplementary Planes）と呼ばれる。

Unicode のコードポイントの値を記憶媒体上で表現するには 1 バイトでは不十分である。従って日本語をはじめとする全角文字を表現するには複数バイトの記憶領域が必要⁶⁰ となる。実際に Unicode のコードポイントの値をどのような形で記憶媒体上のバイト列として符号化するかは「Unicode の**文字符号化方式**」として規定されており、文字符号化方式の具体的なもの（**文字符号化スキーム**：CES）として、UTF-8、UTF-16、UTF-16LE、UTF-16BE、UTF-32、UTF-32LE、UTF-32BE がある。

1 つの文字を記憶媒体上で表現する場合、Unicode の CES 毎にそのサイズが異なる。具体的には、UTF-8 では 1～4 バイトのサイズ（可変長）で、UTF-16 系では 2 もしくは 4 バイト（可変長）、UTF-32 系では 4 バイト固定のサイズで 1 文字のデータが格納される。特に UTF-16 では、1 つの文字をバイト表現する際、上位 2 バイトを**上位サロゲート**（High Surrogate）、下位 2 バイトを**下位サロゲート**（Low Surrogate）と呼ぶ（当該文字が 2 バイト表現の場合は下位サロゲートはない⁶¹）。また UTF-16 において、1 つの文字が上下サロゲートを持つ 4 バイトで表現される場合、それを**サロゲートペア**と呼ぶ。

Python 言語では Unicode の CES をはじめ様々なエンコーディングに対応している。

ここで述べたような Unicode に関する各種規定（コードポイントの範囲や符号化方式、符号化スキームなど）は **Unicode の符号化モデル**と呼ばれるものであるが、これの詳細に関する説明は割愛する。詳しくは Unicode の公式インターネットサイト The Unicode Consortium (<http://www.unicode.org/>) で公開されている情報を参照のこと。

■ string モジュールに定義されている文字の種別

string モジュールには表 10 に示すような文字の種別の定義がある。

例. 表 10 の定義の一部を閲覧する

```
>>> import string Enter ← string モジュールの読み込み
>>> string.ascii_lowercase Enter ← ascii_lowercase を表示
'abcdefghijklmnopqrstuvwxyz' ← ascii_lowercase の内容
>>> string.digits Enter ← digits を表示
'0123456789' ← digits の内容
```

⁶⁰このことにより、全角文字は**マルチバイト文字**（多バイト文字）であると言われる。

⁶¹UCS-2 の文字符号化方式。

表 10: string モジュールで定義されている半角文字の種別 (一部)

定義	内容	定義	内容
<code>ascii_lowercase</code>	英小文字	<code>ascii_uppercase</code>	英大文字
<code>ascii_letters</code>	<code>ascii_lowercase</code> と <code>ascii_uppercase</code> を合わせたもの	<code>digits</code>	半角数字
<code>hexdigits</code>	16 進数の表記を構成する文字	<code>octdigits</code>	8 進数の表記を構成する文字
<code>printable</code>	印字可能文字	<code>whitespace</code>	空白文字

表 10 の定義を利用すると、半角文字の種別の判定ができる。

例. 半角文字の種別の判定

```
>>> 'a' in string.ascii_lowercase Enter ←'a' は小文字か?
True ←真
>>> 'A' in string.ascii_lowercase Enter ←'A' は小文字か?
False ←偽
```

参考) ここで説明したことに加え、後の「2.7.1.1 出力データの書式設定」(p.103) で説明する **フォーマット済み文字列リテラル** (f-string) を利用すると、更に高度な文字列編集が可能となる。

2.4.7.15 文字列のデータサイズについて

Python の文字列 (str) は、扱う文字の種別によってシステム内部での表現が異なるため、メモリ上でのデータサイズが文字数と同じにはならない (表 11 参照)。

表 11: システム内部での文字列のデータサイズ

文 字 列	データサイズ (単位: バイト)	備 考
1) ASCII 文字のみから成る文字列	文字数 + 管理情報のサイズ	
2) 基本多言語面 (BMP) の文字と ASCII 文字のみから成る文字列	文字数 × 2 + 管理情報のサイズ	UCS-2 ベース
3) 補助平面の文字を含む文字列	文字数 × 4 + 管理情報のサイズ	UTF-32 ベース

このことは次に示すプログラム `strSize01.py` で確認することができる。

プログラム: strSize01.py

```
1 import sys
2
3 s0 = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890***'
4 s1 = 'きょうはよいひでしたね。あすもよいひならいいですね。ごきげんよう。'
5 s2 = 'きょうはよいひでしたね。あすもよいひならいいですね。ごきげんよう\U0001f600'
6
7 print('s0: ', s0, sep='')
8 print('s1: ', s1, sep='')
9 print('s2: ', s2, '\n', sep='')
10
11 print('s0: length=', len(s0), ', data size=', sys.getsizeof(s0), sep='')
12 print('s1: length=', len(s1), ', data size=', sys.getsizeof(s1), sep='')
13 print('s2: length=', len(s2), ', data size=', sys.getsizeof(s2), sep='')
```

このプログラムの `s0`, `s1`, `s2` はそれぞれ、ASCII 文字のみから成る文字列、BMP のみから成る文字列、補助平面の文字を含む文字列であり、実行すると、次のような出力が得られる。

```
s0: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890***
s1: きょうはよいひでしたね。あすもよいひならいいですね。ごきげんよう。
s2: きょうはよいひでしたね。あすもよいひならいいですね。ごきげんよう😄

s0: length=66, data size=107
s1: length=33, data size=124
s2: length=33, data size=192
```

この出力から、`s1`, `s2` の文字数が同じであるにもかかわらず、データサイズが大きく異なることがわかる。

参考) プログラム strSize01.py では、Python 処理系上でのオブジェクトのメモリサイズを調べるために sys.getsizeof を使用している。

書き方: sys.getsizeof(オブジェクト)

「オブジェクト」のサイズ(単位:バイト)を返す。ただしこの値は、複雑なデータ構造の場合は全要素を含めたメモリサイズではないことに注意すること。

2.4.8 真理値 (bool 型)

真理値 (bool 型) は真か偽かを現す値で、True, False の2値から成る⁶²。例えば、両辺の値が等しいかどうかを検査する比較演算子 '==' があり、1 == 2 という式は誤りなのでこの式の値は False となる。(次の例を参照)

例. 判定結果を変数に代入する

```
>>> p = 1==2  Enter    ← 1==2 の検査結果を変数 p に与えている。
>>> print( p )  Enter    ← 変数 p の内容を表示する。
False           ← 変数 p の内容が False (偽) であることがわかる。
```

このように、値の比較をはじめとする条件検査の式は真理値を与えるものであり、「2.6 制御構造」(p.89) のところで条件判定の方法として解説する。

参考) 真理値と数値の間の演算

数値の計算に真理値を使用すると、True は 1, False は 0 として扱われる。

例. 真理値と数の演算

```
>>> 1.2 + True  Enter
2.2             ← True は 1 とみなされる
>>> False * 5.0 Enter
0.0             ← False は 0 とみなされる
```

2.4.9 ヌルオブジェクト: None

ヌルオブジェクトは「値を持たない」ことを意味する特殊なオブジェクトであり、None と記述する。

例. ヌルオブジェクト

```
>>> a  Enter    ← 未設定の記号 (変数) a を参照するとエラーが発生する
Traceback (most recent call last):    ← 以下、エラーメッセージ
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> a = None  Enter    ← a にヌルオブジェクトを設定
>>> a  Enter    ← 内容確認
>>>          ← 何も表示されない (しかし、a はヌルオブジェクトを保持している)
```

None は print 関数で出力すると「None」と出力される。

例. print 関数による None の出力 (先の例の続き)

```
>>> print( a )  Enter
None
```

None の型は NoneType である。

例. None の型 (先の例の続き)

```
>>> type( a )  Enter    ← 型の検査 (p.52 「2.4.12 型の検査」で解説する)
<class 'NoneType'>      ← NoneType
```

NoneType 型のオブジェクトは唯一つ None のみである。

参考) ある型 (クラス) が唯一つのオブジェクトを持つ場合、そのオブジェクトを**シングルトン**であるという。

例. 「None は NoneType クラスのシングルトンオブジェクトである」

⁶²bool は int のサブクラスである。詳しくは「4.30 Python の型システム」(p.362)を参照のこと。

2.4.10 式や値の記述の省略

実際のプログラム開発においては、式や値の記述を一時的に棚上げして省略し、後で具体的な式や値を記述することがしばしばある。そのような場合に、式や値の代わりに Ellipsis '...' を用いる。

例. '...' の記述

```
>>> x = ... Enter    ←変数 x に代入する値を一時的に棚上げして省略
>>>                                ←とりあえず代入処理は終わる
```

ただし、これは式や値の記述の一時的な保留であり、実際の処理においては上記 x に意味のある値を代入する記述に修正しなければならない。(下の例)

例. 変数 x の内容 (先の例の続き)

```
>>> x Enter    ←内容確認
Ellipsis      ←このような値
>>> type( x ) Enter    ←型を調べる
<class 'ellipsis'>    ←このような型
```

Ellipsis '...' は ellipsis クラスのシングルトンオブジェクトである。

2.4.11 型の変換

代表的な値の型として数値、文字列、真理値があるが、これら異なる型の間でデータを変換するには、

型 (値)

とする⁶³。型の変換処理の例を表 12 に挙げる。

表 12: 型の変換の例

変換元のデータ	整数へ	浮動小数点数へ	文字列へ	真理値へ
整数 浮動小数点数	- int(3.1) → 3	float(3) → 3.0 -	str(3) → '3' str(3.1) → '3.1'	bool(3) → True bool(3.1) → True 0 や 0.0 は False となる
文字列	int('2') → 2	float('2.1') → 2.1 -	-	bool('a') → True bool('True') → True bool('False') → True (注意!) 空文字列 '' は False となる
真理値	int(True) → 1 int(False) → 0	float(True) → 1.0 float(False) → 0.0	str(True) → 'True' str(False) → 'False'	- -

2.4.11.1 各種の値の文字列への変換 (str, repr)

各種の型の値を文字列に変換する⁶⁴ には表 12 で示した str を使用する。

例. int, float の値を文字列に変換する

```
>>> str( 2 ) Enter    ←整数の 2 を文字列に変換
'2'          ←変換結果
>>> str( 3.14 ) Enter    ←浮動小数点数の 3.14 を文字列に変換
'3.14'       ←変換結果
```

また、str とは別に repr もある。

例. repr による文字列への変換

```
>>> repr( 2 ) Enter    ←整数の 2 を文字列に変換
'2'          ←変換結果
>>> repr( 3.14 ) Enter    ←浮動小数点数の 3.14 を文字列に変換
'3.14'       ←変換結果
```

⁶³この方法は、クラスのコンストラクタによってオブジェクトを生成する機能によるものである。詳しくは「2.9 オブジェクト指向プログラミング」(p.154) で解説する。

⁶⁴データをファイルに出力する場合や通信路に送出する場合などにおいて多用する処理である。

一見すると repr は str と同じ処理をするかのように見えるが、より複雑なオブジェクトを文字列に変換する場合にその違いがわかる。次の例は mpmath の mpf オブジェクトを文字列に変換するものである。

例. mpmath の mpf オブジェクトを文字列に変換する

```
>>> from mpmath import mp      Enter      ←モジュールの読み込み
>>> mp.dps = 60                 Enter      ←演算精度の設定
>>> x = mp.factorial(30)        Enter      ← mpf オブジェクトによる 30! の計算
>>> str( x )                    Enter      ← str による文字列への変換
'265252859812191058636308480000000.0'    ←変換結果
>>> repr( x )                   Enter      ← repr による文字列への変換
"mpf('265252859812191058636308480000000.0')" ←変換結果
```

このように、repr は元のデータ型がわかる形で文字列に変換する。

2.4.12 型の検査

type 関数を用いるとデータの型を調べることができる。

例. type 関数による型の検査

```
>>> type( 2 )                  Enter      ←整数値の型を調べる
<class 'int'>                  ←「整数」を表す型
>>> type( 3.14 )               Enter      ←浮動小数点数の型を調べる
<class 'float'>                ←「浮動小数点数」を表す型
>>> type('abc' )               Enter      ←文字列の型を調べる
<class 'str'>                  ←「文字列」を表す型
>>> type( [1,2,3] )             Enter      ←リストの型を調べる
<class 'list'>                 ←「リスト」65を表す型
```

この例が示すように、'<class 型名>' という形式 (type 型オブジェクト) で型に関する情報が得られる。また、type 型オブジェクトの __name__ 属性から型名を表す文字列が得られる。(次の例)

例. 文字列形式で型名を取得する

```
>>> a = type( 2 )              Enter      ← type 型オブジェクトの取得
>>> a.__name__                  Enter      ←型名の文字列を取得
'int'                            ←文字列として得られた
```

データの型の検査は is 演算子⁶⁶ を用いて

type(値) is 型名

という形で判定することも可能⁶⁷ であり、判定結果が真理値として得られる。(次の例)

例. type 関数と is 演算子による型の検査

```
>>> type( 2 ) is int           Enter      ←整数値「2」が int 型かどうかの検査
True                            ←真
>>> type( 2 ) is float         Enter      ←整数値「2」が float 型かどうかの検査
False                           ←偽
```

この例において、is の後ろに記述された int, float は type 型オブジェクトとして予約されている。他の型についても試されたい。

型の検査には isinstance 関数を用いる方法もある。

書き方: isinstance(オブジェクト, 型名)

「オブジェクト」の型 (クラス) が「型名」の場合に True を、そうでない場合に False を返す。

⁶⁵リストに関しては「2.5.1 リスト」(p.57) で説明する。

⁶⁶is に関しては後の「2.6.4.5 is 演算子による比較」(p.101) で説明する。

⁶⁷この方法で多くの種類のオブジェクトの型を判定可能であるが、「型名」として指定できないものもあるので、各種のケースについて試していただきたい。

例. isinstance 関数による型の検査

```
>>> isinstance( 3, int )  Enter    ← 3 は int 型か
True                      ← 真: int 型である
>>> isinstance( 3.14, int ) Enter    ← 3.14 は int 型か
False                     ← 偽: int 型ではない
```

2.4.12.1 float の値が整数値かどうかを検査する方法

float の値が整数値 (小数点以下が 0 である値) かどうかを判定するには is_integer メソッドを使用する。

例. is_integer メソッドによる判定 (真になる場合)

```
>>> a = 3.0  Enter    ← float 型の 3.0 を変数 a に設定
>>> type(a)  Enter    ← 型の検査
<class 'float'>      ← float 型である
>>> a.is_integer() Enter    ← 値が整数値かどうか検査
True                 ← 整数値である
```

例. is_integer メソッドによる判定 (偽になる場合)

```
>>> a = 3.14  Enter    ← float 型の 3.14 を変数 a に設定
>>> type(a)  Enter    ← 型の検査
<class 'float'>      ← float 型である
>>> a.is_integer() Enter    ← 値が整数値かどうか検査
False                ← 整数値ではない
```

このように判定結果が真理値で得られる。

注) Python 3.11 の版までは、int 型のオブジェクトに対しては is_integer メソッドは定義されておらず、使用することができない。

例. Python 3.11 での int 型変数に対する is_integer メソッドの試み (Windows 環境)

```
C:\Users\katsu>py -3.11    ← Python 3.11 を起動
Python 3.11.9 (tags/v3.11.9:de54cf5, Apr 2 2024, 10:12:12) [MSC v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3  Enter    ← int 型の 3 を変数 a に設定
>>> type(a) Enter    ← 型の検査
<class 'int'>      ← int 型である
>>> a.is_integer() Enter    ← is_integer を試みると…
Traceback (most recent call last):      ← そのようなメソッドは使えないという旨の
  File "<stdin>", line 1, in <module>    エラーメッセージが表示される
AttributeError: 'int' object has no attribute 'is_integer'
```

例. Python 3.12 以降では int 型変数に対する is_integer メソッドが有効 (Windows 環境)

```
C:\Users\katsu>py -3.12    ← Python 3.12 を起動
Python 3.12.7 (tags/v3.12.7:0b05ead, Oct 1 2024, 03:06:41) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3  Enter    ← int 型の 3 を変数 a に設定
>>> type(a) Enter    ← 型の検査
<class 'int'>      ← int 型である
>>> a.is_integer() Enter    ← is_integer を試みると…
True               ← 結果が得られる
```

2.4.13 基数の変換

2.4.13.1 n 進数 → 10 進数

int 関数⁶⁸ の第 2 引数に基数を指定すると、文字列として記述された10進数以外の表現の数値を整数値に変換できる。

⁶⁸正確には int コンストラクタ。

例. 2 進数→10 進数の変換

```
>>> int('1111',2) Enter ← 2 進数の '1111' を 10 進数に変換
15 ← 変換結果 (整数)
>>> int('0b1111',2) Enter ← 2 進数の接頭辞 '0b' (ゼロビー) も使用可能
15 ← 変換結果 (整数)
```

例. 8 進数→10 進数の変換

```
>>> int('77',8) Enter ← 8 進数の '77' を 10 進数に変換
63 ← 変換結果 (整数)
>>> int('0o77',8) Enter ← 8 進数の接頭辞 '0o' (ゼロオー) も使用可能
63 ← 変換結果 (整数)
```

例. 16 進数→10 進数の変換

```
>>> int('FF',16) Enter ← 16 進数の 'FF' を 10 進数に変換
255 ← 変換結果 (整数)
>>> int('0xFF',16) Enter ← 16 進数の接頭辞 '0x' (ゼロエックス) も使用可能
255 ← 変換結果 (整数)
```

2.4.13.2 10 進数→n 進数

10 進数の整数値を別の基数表現の文字列に変換する次のような関数がある。

働き	10 進数→2 進数	10 進数→8 進数	10 進数→16 進数
関数名	bin	oct	hex

これら関数は第 1 引数に整数値を取る。

例. 10 進数を他の基数表現に変換する

```
>>> bin(15) Enter ← 15 を 2 進数表現の文字列に変換
'0b1111' ← 変換結果 (文字列)
>>> oct(15) Enter ← 15 を 8 進数表現の文字列に変換
'0o17' ← 変換結果 (文字列)
>>> hex(15) Enter ← 15 を 16 進数表現の文字列に変換
'0xf' ← 変換結果 (文字列)
```

参考) 浮動小数点数 (float 型) から 2 進数への変換とその逆の変換を行うサンプルプログラムを巻末付録「I.5 浮動小数点数と 2 進数の間の変換」(p.462) で示す。

参考) mpmath ライブラリを用いると高い精度で 2 進数の表現を得ることができる。これに関しては p.27 「【参考】浮動小数点数の 2 進数表現」を参照のこと。

2.4.14 ビット演算

表 13 に整数のビット演算の書き方を示す。

表 13: ビット演算のための演算子

演算	説明	演算	説明	演算	説明
$x \& y$	x と y の論理積	$x y$	x と y の論理和	$x \wedge y$	x と y の排他的論理和
$x \ll n$	x を左に n ビットシフト	$x \gg n$	x を右に n ビットシフト	$\sim x$	x をビット反転 (1 の補数)

注) ここで言う論理積, 論理和, 排他的論理和はビット毎の演算を意味する。

ビット演算は桁の長い整数に対しても実行できる。

例. ビット毎の論理積

```
>>> a = int('11111111',2) Enter ← 2 進数表現で値を生成 (1)
>>> b = int('11110000',2) Enter ← 2 進数表現で値を生成 (2)
>>> c = a & b Enter ← 上記 2 つの数の論理積
>>> print( bin(c) ) Enter ← 2 進数表現に変換
0b11110000 ← 結果表示
```

例. ビット毎の論理和

```
>>> a = int('00001111',2)  Enter  ← 2進数表現で値を生成 (1)
>>> b = int('11110000',2)  Enter  ← 2進数表現で値を生成 (2)
>>> c = a | b  Enter  ← 上記 2 つの数の論理和
>>> print( bin(c) )  Enter  ← 2進数表現に変換
0b11111111  ← 結果表示
```

例. ビット毎の排他的論理和

```
>>> a = int('11111111',2)  Enter  ← 2進数表現で値を生成 (1)
>>> b = int('10101010',2)  Enter  ← 2進数表現で値を生成 (2)
>>> c = a ^ b  Enter  ← 上記 2 つの数の排他的論理和
>>> print( bin(c) )  Enter  ← 2進数表現に変換
0b1010101  ← 結果表示
```

例. 左に n ビットシフト (2^n 倍)

```
>>> a = int('1',2)  Enter  ← 2進数表現で値を生成
>>> print( a << 1 )  Enter  ← 左に 1 ビットシフト (2 倍)
2  ← 結果表示
>>> print( a << 2 )  Enter  ← 左に 2 ビットシフト (4 倍)
4  ← 結果表示
>>> print( a << 3 )  Enter  ← 左に 3 ビットシフト (8 倍)
8  ← 結果表示
```

例. 右に n ビットシフト ($1/2^n$ 倍)

```
>>> a = int('1000',2)  Enter  ← 2進数表現で値を生成
>>> print( a >> 1 )  Enter  ← 右に 1 ビットシフト (1/2 倍)
4  ← 結果表示
>>> print( a >> 2 )  Enter  ← 右に 2 ビットシフト (1/4 倍)
2  ← 結果表示
>>> print( a >> 3 )  Enter  ← 右に 3 ビットシフト (1/8 倍)
1  ← 結果表示
```

例. ビット反転 (1 の補数)

```
>>> a = int('1',2); b = ~a; print( bin(b) )  Enter  ← 2進数表現の '1' をビット反転
-0b10  ← 結果表示
>>> a = int('11',2); b = ~a; print( bin(b) )  Enter  ← 2進数表現の '11' をビット反転
-0b100  ← 結果表示
>>> a = int('111',2); b = ~a; print( bin(b) )  Enter  ← 2進数表現の '111' をビット反転
-0b1000  ← 結果表示
```

‘~’によるビット反転は、計算結果と元の値の和が -1 となる形で実行される。

課題) `int('0',2)` や `int('00',2)` を ‘~’ でビット反転するとどのような結果になるかを確認せよ。

また、何故そのような結果となるか考察せよ。

2.4.14.1 ビット演算の累算代入

変数に格納された値そのものを変更する形でビット演算を実行することができる。(下記参照)

`x &= y` は `x = x & y` と同じ, `x |= y` は `x = x | y` と同じ, `x ^= y` は `x = x ^ y` と同じ,
`x <<= y` は `x = x << y` と同じ, `x >>= y` は `x = x >> y` と同じ,

2.4.14.2 整数値のビット長を求める方法

与えられた整数値を表現するのに最低限必要なビット長を調べるには `bit_length` メソッドを使用する。

例. 1023 を表現するのに最低限必要なビット数を求める

```
>>> a = 1023  [Enter]    ← int 型の 1023 を変数 a に設定
>>> a.bit_length() [Enter] ← ビット長を調べる
10              ← 10 ビットの長さ
>>> bin(a) [Enter]    ← 実際に 1023 を 2 進数に変換して確認する
'0b1111111111'    ← 2 進数で 10 桁 (10 ビットの長さ)
```

注) bit_length メソッドは float 型に対しては使用できない。

2.4.15 バイト列

計算機の記憶資源上のバイトデータの並びを表現するためのデータ型として bytes 型がある。様々なデータを入出力や通信に使用する際に、対象のデータをこの型のオブジェクトに変換して取り扱うことが多い。

bytes 型のデータは先頭に b を伴う引用符で括る形式で表現され、引用符の中は 16 進数で表記する。

例. bytes 型の表現

```
>>> b = b'\x01\xff' [Enter]    ← bytes 型のデータの表記
>>> b [Enter]        ← 内容確認
b'\x01\xff'
```

このように bytes 型の値は、1 バイトずつの 16 進数 00~ff の先頭に \ を付けて書き並べる。

整数値をバイト列に変換するには to_bytes メソッドを使用する。

書き方： 整数値.to_bytes(バイト長, バイトオーダー)

「整数値」を指定した「バイトオーダー」⁶⁹ の順 ('big' もしくは 'little') で「バイト長」の長さの bytes オブジェクトに変換して返す。

例. 整数値を bytes オブジェクトに変換する

```
>>> a = 2882400000 [Enter]    ← この整数値を
>>> b = a.to_bytes(4, 'big') [Enter]    ← 4 バイトのバイト列に変換する
>>> b [Enter]            ← 確認
b'\xab\xcd\xef\x00'    ← このようなバイト列になっている
```

念のため、この例で用いた a の値を hex 関数で確認する。

例. hex 関数による確認 (先の例の続き)

```
>>> hex(a) [Enter]
'0xabcdef00'    ← 先の例と同じ 16 進値になっている
```

バイト列を整数値に変換するには int.from_bytes を使用する。

書き方： int.from_bytes(バイト列, バイトオーダー)

指定した「バイトオーダー」で「バイト列」を整数値に変換して返す。

例. バイト列を整数値に変換する (先の例の続き)

```
>>> int.from_bytes(b, 'big') [Enter]
2882400000    ← 先の a の値と同じ
```

文字列をバイト列に変換する方法については後の「2.7.3.4 バイト列の扱い」(p.112) で解説する。また、バイト列の内容を様々なデータ形式として読み取る方法については「4.11.2 バイナリデータの展開」(p.314) で解説する。

bytes 型のオブジェクトは事後に変更不可能⁷⁰ である。そのため、編集可能なバイト列として bytearray 型がある。これに関しては「4.13 編集可能なバイト列： bytearray」(p.317) で解説する。

⁶⁹ バイトオーダーに関しては後の「4.11.3 バイトオーダーについて」(p.314) で解説する。

⁷⁰ イミュータブルであるという。「イミュータブル」については後の「ミュータブル／イミュータブル」(p.69) で解説する。

2.5 データ構造

ここでは、複数のデータ要素を保持する**データ構造**について解説する。具体的にはリスト、タプル、セット、辞書（それらをまとめて**コンテナ**と呼ぶ）について解説する。

2.5.1 リスト

複数のデータの並び（要素の並び）を**リスト**として表すことができる。要素の区切りにはコンマ「,」を使用する。例えば、0～9 の数を順番に並べたものはリストとして

```
[0,1,2,3,4,5,6,7,8,9]
```

と表すことができる。このようにリストは「[」と「]」で括ったデータ構造である。リストの要素としては任意の型のデータを並べることができる。例えば、

```
['nakamura',51,'tanaka',22,'hashimoto',14]
```

のように、型の異なるデータが要素として混在するリストも作成可能である。また、リストを要素として持つリストも作成可能であり、例えば、

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

といったリストも作成することができる。

要素を持つリストの末尾には余分なコンマが1つ存在しても良い。

例. 末尾の余分なコンマ

```
>>> [1,2,3] Enter ←要素を持つリスト
[1, 2, 3] ←問題ない
>>> [1,2,3,] Enter ←末尾に余分なコンマが1つある場合は
[1, 2, 3] ←余分なコンマは無視される
>>> [1,2,3,,] Enter ←末尾に余分なコンマが複数ある場合は
File "<stdin>", line 1 ←エラーとなる
[1,2,3,,]
SyntaxError: invalid syntax
```

2.5.1.1 空リスト

要素を持たない**空リスト**は「[]」と記述する。これはヌルオブジェクト `None` とは異なる。空リストは `list` 関数⁷¹ に引数を与えずに「`list()`」として評価することでも得られる。

2.5.1.2 リストの要素へのアクセス

リストの要素を取り出すには、取り出す要素の位置を示すインデックスを「[]」で括って⁷² 指定する。これは次の例を見ると理解できる。

例. スライスの指定による要素の参照

```
>>> lst = [2,4,6,8,10] Enter ←リストの生成
>>> lst[1] Enter ←1番目の要素を指定
4 ←対象の要素が表示されている
```

このようにリストの要素は**0番目から始まる位置**（インデックス）を指定してアクセスすることができる。リストの中の、インデックスで指定した特定の範囲（部分リスト）を取り出すこともできる。

例. 部分リストの取り出し

```
>>> lst = [2,4,6,8,10] Enter ←リストの生成
>>> lst[1:4] Enter ←部分リスト（1～4-1番目）の取り出し
[4, 6, 8] ←部分リストが得られている
```

この例のように、コロン「:」でインデックスの範囲を指定するが、 $[n_1:n_2]$ と指定した場合は、 $n_1 \sim (n_2 - 1)$ の範囲の部分を示していることに注意しなければならない。

⁷¹正確には `list` コンストラクタという。

⁷²「[...]」の部分のスライスと呼び、文字列をはじめとする多くのデータ構造で同様の扱いができる。

リストの要素にアクセスするためのスライスにはインデックスの増分を指定することができる。

例. 飛々のインデックス指定による要素の取り出し

```
>>> lst = [0,1,2,3,4,5,6,7]  Enter  ←リストの生成
>>> lst[1:6:2]  Enter  ←飛々のインデックス指定
[1, 3, 5]  ←得られたリスト
```

この例のようにスライスを $[n_1:n_2:n_3]$ と記述して n_3 にインデックスの増分を与えることができる。これに関しては後の「2.5.5.4 不連続な部分の取り出し」(p.84) で更に詳しく説明する。

リストを用いることで、複数の要素を持つ複雑な構造を1つのオブジェクトとして扱うことができる。この意味で、リストは複雑な情報処理を実現するに当たって非常に有用なデータ構造である。

2.5.1.3 リストの編集

リストは要素の追加や削除を始めとする編集ができるデータ構造である。リストを編集するための基本的な機能について解説する。

● 要素の書き換え

リストの指定したインデックスの要素を直接書き換えることができる。

例. リストの要素の書き換え (1)

```
>>> lst = [0,1,2,3,4,5]  Enter  ←リストの作成
>>> lst[3] = 'x'  Enter  ←3番目の要素を'x'に変更
>>> lst  Enter  ←内容確認
[0, 1, 2, 'x', 4, 5]  ←要素が変更されている
```

例. リストの要素の書き換え (2) (先の例の続き)

```
>>> lst[2:4] = ['a','b']  Enter  ←2~3番目の要素を書き換える
>>> lst  Enter  ←内容確認
[0, 1, 'a', 'b', 4, 5]  ←変更結果
```

代入の左辺に指定した要素範囲よりも長い(要素数の多い)リストを右辺に与えても無事に代入処理が完了する。その場合は元のリストが拡張される。(試されたい)

● 要素の追加

リストに対して要素を末尾に追加するには `append` メソッドを使用する。

例. リストへの要素の追加

```
>>> lst = ['x','y']  Enter  ←リストの作成
>>> lst.append('z')  Enter  ←要素の追加
>>> lst  Enter  ←内容確認
['x', 'y', 'z']  ←要素が追加されている
```

これは、リスト `lst` の末尾に要素 `'z'` を追加している例である。

《 オブジェクトに対するメソッドの適用 》

Python ではオブジェクト指向の考え方に則って、

作用対象オブジェクト.メソッド

というドット `'.'` を用いた書き方で、メソッドをオブジェクトに対して適用する。

先の例ではリスト `lst` に対して、メソッド `append('z')` を適用している。

《 リストへの要素の追加 》

書き方: 対象リスト.append(追加する要素)

「対象リスト」そのものが変更される..

● 要素の挿入

リストの指定した位置に要素を挿入するには insert メソッドを使用する。

例. リストの指定した位置に要素を挿入

```
>>> lst = ['a','c','d']  Enter    ←リストの作成
>>> lst.insert(1,'b')    Enter    ←要素の挿入
>>> lst                  Enter    ←内容確認
['a', 'b', 'c', 'd']      ←要素が挿入されている
```

これは、リスト lst の 1 番目に要素 'b' を挿入している例である。

● リストの連結 (1)

演算子 '+' によって複数のリストを連結し、結果を新しいリストとして作成することができる。

例. リストの連結

```
>>> lst1 = ['a','b']      Enter    ←リストの作成 (1)
>>> lst2 = ['c','d']      Enter    ←リストの作成 (2)
>>> lst3 = ['e','f']      Enter    ←リストの作成 (3)
>>> lst4 = lst1 + lst2 + lst3 Enter    ←リストの連結
>>> lst4                  Enter    ←内容確認
['a', 'b', 'c', 'd', 'e', 'f']      ←3つのリストが連結されている
```

'+' によるリストの連結処理において重要なことに、連結結果が**新たなリストとして生成されている**ということがある。すなわち、この例においてリスト lst1, lst2, lst3 の内容は処理の前後で変更はなく、連結結果のリストが新たなリスト lst4 として生成されている。また、累算代入「+=」でリストを追加連結することも可能であるが、その場合は連結の前後でリストは同一のオブジェクト（同じ識別値）である。

例. 累算代入によるリストの連結

```
>>> lst = ['a','b','c']   Enter    ←リストの作成
>>> lst += ['d','e','f']   Enter    ←リストの連結
>>> lst                   Enter    ←内容確認
['a', 'b', 'c', 'd', 'e', 'f']      ←lst が拡張されている
```

これはリストオブジェクト lst を拡張する例である。同様の処理を次に解説する extend メソッドで実行することもできる。

● リストの連結 (2)

'+' によるリストの連結処理とは別に、与えられたリストを編集する形でリストを連結することも可能である。

《 extend メソッドによるリストの連結 》 - リストの拡張 -

書き方： 対象リスト.extend(追加リスト)

「対象リスト」の末尾に追加リストを連結する。結果として対象リストそのものが拡張される。

例. リストの拡張

```
>>> lst = ['a','b','c']   Enter    ←リストの作成
>>> lst.extend(['d','e','f']) Enter    ←リストの拡張
>>> lst                   Enter    ←内容の確認
['a', 'b', 'c', 'd', 'e', 'f']      Enter    ←リストが拡張されている
```

● リストの繰り返し

※ 演算子によってリストの繰り返しを取得することができる。

例. リストの繰り返し

```
>>> lst = [1,2,3]         Enter    ←このリストを
>>> lst * 3               Enter    ←3回繰り返す
[1, 2, 3, 1, 2, 3, 1, 2, 3]      ←処理結果
```

▲注意▲ リストの多重な繰り返し

リストの多重な繰り返しについては注意すべきことがある。例えば次のような繰り返しで作成されたリストについて考える。

例. 二重の繰り返しによるリスト

```
>>> lst = [[0]*4]*3  Enter  ←「繰り返し」の繰り返し
>>> lst  Enter  ←内容確認
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

このような lst が作成されることは理解し易いが、次の処理結果に注意すること。

例. リストの要素の変更（先の例の続き）

```
>>> lst[1][2] = 3  Enter  ←指定した位置の要素を変更
>>> lst  Enter  ←内容確認
[[0, 0, 3, 0], [0, 0, 3, 0], [0, 0, 3, 0]]  ← 3 箇所の要素が変更されている！
```

これに対して、次の例では自然な形で値が変更される。

例. 指定位置のみ値が変更される例

```
>>> lst2 = [[0]*4] + [[0]*4] + [[0]*4]  Enter  ←繰り返したものを3つ連結
>>> lst2  Enter  ←内容確認
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]  ←先の例の場合と同じに見える
>>> lst2[1][2] = 3  Enter  ←指定した位置の要素を変更
>>> lst2  Enter  ←内容確認
[[0, 0, 0, 0], [0, 0, 3, 0], [0, 0, 0, 0]]  ←指定した位置のみが変更されている！
```

これは混乱を招きやすい例である。はじめの例における lst は `[[0]*4]` を3回繰り返したものであり、同一のオブジェクト `[0, 0, 0, 0]` が要素として3つ並んでいるものである。従って、lst 内の要素の1つを変更すると、その影響が他の要素にも表れる。それに対して lst2 では、それぞれ異なるオブジェクトとして `[0, 0, 0, 0]` が3つ要素として並んでおり、1つの `[0, 0, 0, 0]` を変更しても他の要素に影響はない。

このことは些細な注意事項として捉えるべきではなく、

1. 「全く同一のオブジェクト」
2. 「同じ値を持つ別のオブジェクト」

を区別して理解すべきであることが明白になる例である。この件をよく理解するために、後の「2.5.1.12 リストの複製」(p.66)を読んだ後で、再度上の例について考察されたい。

● リストの要素の削除

リストの特定のインデックスの要素を削除するには `del` 文を使用する。

《 del 文によるリストの要素の削除 》

書き方: `del 削除対象のオブジェクト`

削除対象のオブジェクトを削除する。

リストの指定した1つの要素を削除するには次の例のようにする。

例. 指定した要素を削除

```
>>> lst = [2,4,6,8,10]  Enter  ←リストの作成
>>> del lst[1]  Enter  ←リストの1番目の要素の削除
>>> lst  Enter  ←内容確認
[2, 6, 8, 10]  ←リストの要素が削除されている
```

同様の方法で、リストの中の指定した部分を削除することもできる。(次の例)

例. 指定した範囲の削除

```
>>> lst = [2,4,6,8,10]  ←リストの作成
>>> del lst[1:4]  ←リストの1～(4-1) 番目の要素の削除
>>> lst  ←内容確認
[2, 10] ←指定した範囲の要素が削除されている
```

リストから指定した値の要素を取り除くには remove メソッドを使用する.

《 remove メソッドによる要素の削除 》

書き方: 削除対象リスト.remove(値)

削除対象リストの中の指定した値の要素を削除する. 削除対象の値の要素が複数存在する場合は, 最初に見つかった要素を1つ削除する.

例. remove メソッドによる要素の削除

```
>>> lst = ['a','b','c','b','a']  ←リストの作成
>>> lst.remove('b')  ←'b' という値の要素を削除
>>> lst  ←内容確認
['a', 'c', 'b', 'a'] ←最初の'b' が削除されている
```

● 全要素の削除

clear メソッドによってリスト内の全ての要素を削除することができる. この場合, 対象のリストは元のオブジェクトのままである (識別値は変わらない).

例. clear メソッドによる全要素の削除

```
>>> lst = [1,2,3,4]  ←リストの作成
>>> id(lst)  ←識別値の確認
1803804427840 ←lst の識別値
>>> lst.clear()  ←全要素の削除
>>> lst  ←内容確認
[] ←空リストになった
>>> id(lst)  ←lst の識別値の確認
1803804427840 ←元の lst と同じ
```

上記のリスト lst の内容を空にするには, 空リスト [] を lst に代入する方法もあるが, その場合は lst は別のオブジェクト (識別値の異なるオブジェクト) となる.

例. 空リストの代入による全要素の削除 (先の例の続き)

```
>>> lst = []  ←空リストの代入
>>> id(lst)  ←lst の識別値の確認
1803804725952 ←先の識別値と異なる (別のオブジェクトになった)
```

2.5.1.4 リストによるスタック, キューの実現: pop メソッド

リストの特定のインデックスの要素を取り出した後に削除するメソッド pop がある.

《 pop メソッドによる要素の取り出しと削除 》

書き方: 対象リスト.pop(インデックス)

対象リストの中の指定したインデックスの要素を取り出して削除する. このメソッドを実行すると, 削除した要素を返す. また pop の引数を省略すると, 末尾の要素が対象となる.

pop メソッドを用いると, スタック (FILO) やキュー (FIFO) といったデータ構造が簡単に実現できる.

例. pop メソッドによるスタック

```
>>> lst = [] Enter ←空のリストを作成
>>> lst.append('a') Enter ←lstの末尾に要素'a'を追加
>>> lst.append('b') Enter ←lstの末尾に要素'b'を追加
>>> lst.append('c') Enter ←lstの末尾に要素'c'を追加
>>> lst Enter ←lstの内容を確認する
['a', 'b', 'c'] ←要素が蓄積されていることがわかる
>>> lst.pop() Enter ←lstの末尾の要素を取り出す
'c' ←lstの末尾の要素が得られた
>>> lst.pop() Enter ←lstの末尾の要素を取り出す
'b' ←lstの末尾の要素が得られた
>>> lst.pop() Enter ←lstの末尾の要素を取り出す
'a' ←lstの末尾の要素が得られた
>>> lst Enter ←lstの内容を確認
[] ←空になっている
```

参考. 上の例で `lst.pop(0)` として実行するとキューが実現できる. また, 後の「4.16.1 deque」(p.323) で説明する deque を使用すると, 列の回転が簡単に実現できる.

2.5.1.5 リストに対する検査

リストに対する各種の検査方法について説明する.

● 要素の存在検査

リストの中に, 指定した要素が存在するかどうかを検査する `in` 演算子がある.

《 `in` 演算子によるメンバシップ検査 》

書き方: 要素 `in` リスト

リストの中に要素があれば `True`, なければ `False` を返す. 要素が含まれないことを検査するには `not in` と記述する.

例. `in` によるメンバシップ検査

```
>>> lst = ['a', 'b', 'c', 'd'] Enter ←リストの作成
>>> 'c' in lst Enter ←'c'がlstに含まれるか
True ←含まれる
>>> 'e' in lst Enter ←'e'がlstに含まれるか
False ←含まれない
>>> 'e' not in lst Enter ←'e'がlstに含まれないか
True ←含まれない
```

● 要素の位置の特定

リストの中に, 指定した要素が存在する位置を求めるには `index` メソッドを用いる.

《 `index` メソッドによる要素の探索 》

書き方 1: 対象リスト.`index`(要素)

「対象リスト」の中に「要素」が最初に見つかる位置 (インデックス) を求める.

書き方 2: 対象リスト.`index`(要素, p_1 , p_2)

p_1 以上 p_2 未満のインデックス範囲で「要素」が最初に見つかる位置を求める. p_2 を省略すると末尾までが探索の対象となる.

要素が見つからなければ `index` メソッドはエラーとなる.

例. index メソッドによる要素の探索

```
>>> lst = ['a','b','c','d','a','b','c','d']  [Enter]  ←リストの作成
>>> lst.index('c')  [Enter]  ←'c' の位置を求める
2  ←インデックス 2 の位置に最初の 'c' が存在する
```

例. 開始位置を指定して探索

```
>>> lst.index('c',3)  [Enter]  ←インデックス位置 3 以降で 'c' の位置を求める
6  ←インデックス 6 の位置に 'c' が存在する
```

探索する要素が対象リストに含まれていない場合、このメソッドの実行は次の例のようにエラーとなる。

例. 要素が含まれていない場合に起こるエラー

```
>>> lst.index('e')  [Enter]  ←'e' の位置を求める
Traceback (most recent call last):  ←以下、エラーメッセージ
  File "<stdin>", line 1, in <module>
ValueError: 'e' is not in list
```

これは ValueError という種類のエラーである。探索範囲内に目的の要素が見つからない場合も同様のエラーとなる。

例. 指定した範囲内に要素が見つからない場合のエラー

```
>>> lst.index('c',0,2)  [Enter]  ← インデックス 0 以上 2 未満の範囲で 'c' の位置を求める
Traceback (most recent call last):  ←以下、エラーメッセージ
  File "<stdin>", line 1, in <module>
ValueError: 'c' is not in list
```

Python では、エラーが発生する可能性のある処理を行う場合は、適切に**例外処理**の対策をしておくべきである。例えば次のようなサンプルプログラム test03.py のような形にすると良い。

プログラム：test03.py

```
1  # coding: utf-8
2  lst = ['a','b','c','d']
3
4  try:
5      n = lst.index('e')
6      print(n,"番目にあります。")
7  except ValueError:
8      print("要素が見つかりません…")
```

プログラムの 4 行目にある try は、エラー（例外）が発生する可能性のある処理を指定するための文である。すなわち 5,6 行目で例外が発生した場合は except 以下のプログラムに実行が移る。（処理系の動作は中断しない⁷³）

2.5.1.6 例外処理

《 例外処理のための try と except 》

書き方： try:
 (例外を起こす可能性のある処理)
 except エラー（例外）の種類:
 (例外を起こした場合に実行する処理)
 except:
 (上記以外の例外を起こした場合に実行する処理)
 finally:
 (全ての except の処理の後に共通して実行する処理)

except は複数記述することができる。

先のプログラム test03.py を実行すると次のような結果となる。

⁷³更に p.337 「4.19 例外（エラー）の処理」では、システムが生成する例外やエラーに関するメッセージを文字列型データとして取得する方法を紹介する。


```
py test03.py [Enter] ← OS のコマンドからプログラムを実行
要素が見つかりません… ←結果の表示
```

5 行目の index メソッドの引数を様々に変更して実行を試みられたい。例外処理に関して後の「4.19 例外（エラー）の処理」（p.337）で更に詳しく解説する。

2.5.1.7 要素の個数のカウント

リストは同じ要素を複数持つことができるが、指定した要素（値）がリストにいくつ含まれるかをカウントするメソッド count がある。

《 count メソッドによる要素のカウント 》

書き方： 対象リスト.count(要素)

「対象リスト」の中の「要素」の個数を求める。

例. count メソッドによる要素のカウント

```
>>> lst = ['a','b','a','c','a','d'] [Enter] ←リストの作成
>>> lst.count('a') [Enter] ←'a' の個数を求める
3 ← 3 個ある
>>> lst.count('e') [Enter] ←'e' の個数を求める
0 ← 0 個（含まれない）
```

リストの長さ（全要素の個数）を求めるには len 関数を使用する。

《 リストの長さ 》

書き方： len(対象リスト)

対象リストの長さ（全要素の個数）を求める。

例. len 関数によるリストの長さの取得

```
>>> lst = [1,2,3,4,5] [Enter] ←リストの作成
>>> len(lst) [Enter] ←長さの算出
5 ← 5 個の要素を持つ
```

参考. 後の「2.10.3 filter」（p.187）で説明する filter 関数と len 関数を応用すると、条件を満たす要素のカウント⁷⁴が実現できる。サンプルを countif01.py に示す。

プログラム：countif01.py

```
1 # coding: utf-8
2 # 条件付きカウントの例
3
4 # 偶数かどうかを判定する関数の定義
5 def isEven( n ):
6     return n%2 == 0
7
8 # 奇数かどうかを判定する関数の定義
9 def isOdd( n ):
10    return n%2 != 0
11
12 # テストデータのリスト
13 lst = [1,2,3,4,5,6,7,8,9]
14 print( 'データリスト:', lst )
15
16 # 偶数である要素の個数
17 c = len( list( filter( isEven, lst ) ) )
18 print( '偶数である要素の個数:', c )
19
20 # 奇数である要素の個数
21 c = len( list( filter( isOdd, lst ) ) )
```

⁷⁴表計算ソフト Microsoft Excel に備わっている COUNTIF 関数と似た機能

このプログラムでは**条件付きカウント**の実現に「指定した条件を満たす要素を取り出して、その個数を調べる」という方法を採用している。このプログラムを実行すると次のようになる。

```
データリスト: [1, 2, 3, 4, 5, 6, 7, 8, 9]
偶数である要素の個数: 4
奇数である要素の個数: 5
```

プログラム中で `def` 文による関数定義をしているが、これに関しては「2.8 関数の定義」(p.137) で解説する。

課題. 後の「2.6.1.7 `for` を使ったデータ構造の生成 (要素の内包表記)」(p.92) で解説する条件付きの内包表記を使用して `countif01.py` と同等の処理をするプログラム `countif01-2.py` を作れ。

2.5.1.8 要素の合計: `sum` 関数

リストの要素が数値である場合、`sum` 関数を用いることで全要素の合計を求めることができる。

例. 要素の合計

```
>>> lst = [1,2,3,4,5]  Enter  ←リストの作成
>>> sum( lst )  Enter  ←合計処理
15  ←合計の値
```

2.5.1.9 要素の整列 (1): `sort` メソッド

リストの要素を整列するには `sort` メソッドを使用する。

例. リストの要素の整列

```
>>> lst = [8,3,2,7,5,6,9,1,4]  Enter  ←リストの作成
>>> lst.sort()  Enter  ←整列の実行
>>> lst  Enter  ←内容確認
[1, 2, 3, 4, 5, 6, 7, 8, 9]  ←整列されている
```

これは、要素の大小を基準にして並べ替えを実行した例である。`sort` メソッドは対象となるリストそのものを変更する。 順序の比較の基準を明に指定するには、`sort` メソッドにキーワード引数⁷⁵ `'key=比較対象のキーを求める関数の名前'` を与える。例えば次の例について考える。

例. 複雑なリストの並べ替え

```
>>> lst = [ [1,'c'], [2,'b'], [3,'a'] ]  Enter  ←複雑なリスト
>>> lst.sort()  Enter  ←これを単純に整列してみる
>>> lst  Enter  ←内容確認
[[1, 'c'], [2, 'b'], [3, 'a']]  ←変化が見られない
```

この例では整列の処理の後でもリストの要素の順番に変化が見られない。これは、リストの各要素を構造的に（素朴に）比較したことによる。次に、リストの各要素の2番目の要素（インデックスの[1]番目）のアルファベットの部分を基準に整列することを考える。（次の例）

例. 各要素のアルファベットの部分を基準にして整列する（先の例の続き）

```
>>> lst.sort( key = lambda x:x[1] )  Enter  ←インデックスの1番目をキーにする整列処理
>>> lst  Enter  ←内容確認
[[3, 'a'], [2, 'b'], [1, 'c']]  ←整列結果
```

例の中にある記述 `'lambda x:x[1]'` は、与えられた `x` の2番目の要素（インデックスの1番目）を取り出す `lambda` 表現である。この部分にはキーを取り出す関数の名前を与えることもできる。`lambda` と **関数** に関しては「2.8 関数の定義」(p.137)、「2.10.2 `lambda` と関数定義」(p.186) のところで解説する。

並べ替えの順番を逆にする場合は `sort` メソッドにキーワード引数 `'reverse=True'` を与える。（次の例）

⁷⁵詳しくは「2.8 関数の定義」(p.137) で解説する。

例. 逆順の整列

```
>>> lst = [1,2,3,4,5]  ←リストデータの作成
>>> lst.sort( reverse=True )  ←逆順で整列
>>> lst  ←内容確認
[5, 4, 3, 2, 1] ←整列結果
```

2.5.1.10 要素の整列 (2) : sorted 関数

リストの整列結果を別のリストとして取得するには sorted 関数を用いる。

例. リストの要素の整列

```
>>> lst = [8,3,2,7,5,6,9,1,4]  ←リストの作成
>>> sorted( lst )  ←整列の実行
[1, 2, 3, 4, 5, 6, 7, 8, 9] ←整列されている
>>> lst  ←元のリストの内容確認
[8, 3, 2, 7, 5, 6, 9, 1, 4] ←変更されていない
```

この例が示すように, sorted 関数は元のリストを変更しない。

sort メソッドと同様に ‘key=’, ‘reverse=’ といったキーワード引数を使用できる。(次の例参照)

例. ‘key=’, ‘reverse=’ を指定して整列

```
>>> lst = [ [1, 'c'], [2, 'b'], [3, 'a'] ]  ←複雑なリスト
>>> sorted( lst, key = lambda x:x[1] )  ←インデックスの1番目をキーにする整列処理
[[3, 'a'], [2, 'b'], [1, 'c']] ←整列結果

>>> lst = [1,2,3,4,5]  ←リストの作成
>>> sorted( lst, reverse=True )  ←逆順で整列
[5, 4, 3, 2, 1] ←整列結果
```

2.5.1.11 要素の順序の反転

リストの要素の順序を反転するには reverse メソッドを使用する。

例. 要素の順序の反転

```
>>> lst = [5,4,3,2,1]  ←リストの作成
>>> lst.reverse()  ←反転の実行
>>> lst  ←内容確認
[1, 2, 3, 4, 5] ←反転されている
```

反転結果を別のデータ (イテレータ) として取得するには reversed 関数を用いる。

例. reversed 関数による要素の順序の反転 (先の例の続き)

```
>>> list( reversed( lst ) )  ←反転されたイテレータをリストに変換
[5, 4, 3, 2, 1] ←反転されている
>>> lst  ←元のリストの内容確認
[1, 2, 3, 4, 5] ←変更されていない
```

イテレータについては「2.6.1.8 イテレータ」(p.92) で説明する。

2.5.1.12 リストの複製

既存のリストの複製には copy 関数を使用する。リストなどのデータ構造に対する処理は対象のデータそのものを改変するものが多い。処理において元のデータの内容を変更したくない場合は、元のデータの複製を作成し、その複製に対して処理を実行するのが良い。

この関数は copy モジュールに含まれるものであり、使用するには copy モジュールをインポートしておく必要がある。

例. リストの複製

```
>>> import copy      Enter      ← copy モジュールをインポート
>>> lst1 = [8,3,2,7,5,6,9,1,4]  Enter  ← リストの作成
>>> lst2 = copy.copy(lst1)      Enter  ← lst1 の複製を lst2 として作成
>>> lst2.sort()      Enter      ← lst2 を整列
>>> lst2      Enter      ← lst2 の内容を確認
[1, 2, 3, 4, 5, 6, 7, 8, 9]      ← 整列されている
>>> lst1      Enter      ← lst1 の内容を確認
[8, 3, 2, 7, 5, 6, 9, 1, 4]      ← 元のままのリストである
```

ただし、この方法による複製では、要素にリストを持つリストの場合に問題が起こることがある。(次の例)

例. リストを要素に持つリストの複製 (先の例の続き)

```
>>> lst3 = [1,[2,3,4],5,[6,7,8]]  Enter  ← リストを要素に持つリスト
>>> lst4 = copy.copy(lst3)      Enter  ← それを複製する
>>> lst4[0] = 10      Enter  ← 複製の先頭要素(数値)を改変
>>> lst4      Enter      ← 内容確認
[10, [2, 3, 4], 5, [6, 7, 8]]      ← 変更できている
>>> lst3      Enter      ← 元のリストの内容を確認
[1, [2, 3, 4], 5, [6, 7, 8]]      ← 影響なし(元のまま)
>>> lst4[1][0] = 20      Enter  ← 複製の要素の内、リストの部分を改変
>>> lst4      Enter      ← 内容確認
[10, [20, 3, 4], 5, [6, 7, 8]]      ← 変更されている
>>> lst3      Enter      ← しかし、元のリストの内容を確認すると…
[1, [20, 3, 4], 5, [6, 7, 8]]      ← 要素のリストの部分が変更されている
```

この例からわかるように、`copy.copy` による複製では要素のリストまでは複製されず、元のリストの要素を参照していることがわかる。このような複製の作成を**浅いコピー (shallow copy)**という。これに対して、要素のリストまで全て(再帰的に)複製することを**深いコピー (deep copy)**という。

`copy` モジュールの `deepcopy` 関数を用いると深いコピーができる。

例. `deepcopy` 関数による深いコピー (先の例の続き)

```
>>> lst3 = [1,[2,3,4],5,[6,7,8]]  Enter  ← リストを要素に持つリスト
>>> lst4 = copy.deepcopy(lst3)      Enter  ← それを deepcopy で複製する
>>> lst4[0] = 10      Enter  ← 複製の先頭要素(数値)を改変
>>> lst4[1][0] = 20      Enter  ← 複製の要素の内、リストの部分を改変
>>> lst4      Enter      ← 複製側の内容確認
[10, [20, 3, 4], 5, [6, 7, 8]]      ← 変更されている
>>> lst3      Enter      ← 元のリストの内容を確認
[1, [2, 3, 4], 5, [6, 7, 8]]      ← 全く変更なし
```

■ 参考 `copy` メソッドによる浅いコピー

リストに対して `copy` メソッドを使用することでも浅いコピーが作成できる。

例. リストに対する `copy` メソッド

```
>>> lst3 = [1,[2,3,4],5,[6,7,8]]  Enter  ← リストを要素に持つリスト
>>> lst4 = lst3.copy()      Enter  ← それを copy メソッドで複製する
```

この例は、リスト `lst3` の浅いコピー `lst4` を作成するものである。

課題. 上の例において `lst4` が `lst3` の浅いコピーになっていることを確かめよ。

■ 参考 `list` コンストラクタによる浅いコピー

既存のリストを `list` コンストラクタに与えて浅いコピーを作成することができる。

例. list コンストラクタによる浅いコピー

```
>>> lst1 = [1,2,[3,4],5]  Enter    ←リストを作成
>>> lst2 = list(lst1); print(lst2)  Enter    ←上記のリストを複製して内容確認
[1, 2, [3, 4], 5]                ←複製されている
>>> lst2[1] = 'x'  Enter    ←複製側の内容を変更
>>> lst1  Enter    ←元のリストの内容確認
[1, 2, [3, 4], 5]                ←変更なし
>>> lst2  Enter    ←複製側は
[1, 'x', [3, 4], 5]              ←変更されている
```

課題. 上の例において lst2 が lst1 の浅いコピーになっていることを確かめよ.

■ 参考

リストに対するメソッドや関数の内、内容の変更を伴わないものについては多くのものが文字列型 (str 型) のオブジェクトに対しても使用できる。(次の例参照)

例. 文字列に対する各種の処理

```
>>> 'abcdefg'.index('c')  Enter    ← index メソッド
2                            ←実行結果
>>> 'abacad'.count('a')  Enter    ← count メソッド
3                            ←実行結果
>>> sorted('dcgeafb')  Enter    ← sorted 関数
['a', 'b', 'c', 'd', 'e', 'f', 'g'] ←この場合の実行結果はリストとして得られる
```

この例で挙げたメソッドや関数以外についても試していただきたい。

2.5.1.13 リストか否かの判定

あるオブジェクトの型がリストか否かを判定するには type 関数を用いる。

例. リストか否かの判定

```
>>> lst = [1,2,3]  Enter    ←変数 lst にリストを与える
>>> type( lst )  Enter    ← type 関数による型の調査
<class 'list'>        ←リストの型
>>> type( lst ) is list  Enter    ← type 関数と is 演算子による型の判定
True                    ←リストである
```

2.5.1.14 リストの内部に他のリストを展開する方法

'*' を用いてリスト内に他のリストを要素として展開することができる。

例. リスト内に他のリストを展開する

```
>>> lst1 = [3,4,5]  Enter    ←このリストの全要素を
>>> lst2 = [1,2,*lst1,6,7]  Enter    ←別のリスト内に展開する
>>> lst2  Enter    ←内容確認
[1, 2, 3, 4, 5, 6, 7]        ←展開されている
```

同様の方法で、種類の異なるデータ構造 (文字列など) も要素として他のリスト内に展開することができる。

2.5.2 タプル

タプルはリストとよく似た性質を持つ、'(' と ')' で括ったデータ構造である。

タプルの例.

```
(0,1,2,3,4,5,6,7,8,9)
('nakamura',51,'tanaka',22,'hashimoto',14)
('a',('b','c'),'d')
```

次のように、リストとタプルは互いに要素として混在させることができる。

例. リスト、タプルの混在

```
['a', ('b', 'c'), 'd']  
('a', ['b', 'c'], 'd')
```

2.5.2.1 リストとタプルの違い

リストとタプルは表記に用いる括弧の記号が異なるだけではない。リストは先に解説したように、要素の追加や削除など自在に編集ができるが、タプルは1度生成した後は変更ができない。

■ ミュータブル／イミュータブル

リストのように、事後で内容の変更が可能であることを「ミュータブル (mutable) である」と言い、事後で内容を変更できないことを「イミュータブル (immutable) である」と言う。より正確に解説すると「イミュータブルであることは、そのオブジェクトが直接保持している値や参照が変更できないこと」である。

例. ミュータブルなリスト

```
>>> lst = ['a', 'b', 'c'] Enter ←リストの作成  
>>> lst[1] = 'z' Enter ←要素の変更  
>>> lst Enter ←内容確認  
['a', 'z', 'c'] ←変更されている
```

この例ではリストの要素を変更している。同様の処理をタプルに対して試みるとエラー (TypeError) が発生する。

例. イミュータブルなタプル

```
>>> tpl = ('a', 'b', 'c') Enter ←タプルの作成  
>>> tpl[1] = 'z' Enter ←要素の変更を試みると…  
Traceback (most recent call last): ←エラーとなる  
File "<stdin>", line 1, in <module>  
    tpl[1] = 'z'  
~~~~~  
TypeError: 'tuple' object does not support item assignment
```

ただし、タプルの要素がリストなどのミュータブルなものである場合、その要素の内容は変更することができる。

例. タプルの要素であるリストの変更

```
>>> tpl = ('a', [1, 2, 3], 'c') Enter ←リストの要素を持つタプル  
>>> tpl[1][1] = 'b' Enter ←リストの部分の内部 (要素) を変更  
>>> tpl Enter ←内容確認  
('a', [1, 'b', 3], 'c') ←リストの部分が変更されている
```

この例が示すことは、タプルがイミュータブルであることに反するような印象を与えるが、上記 `tpl[1]` 自体は変更の前後でその実体が変わっていない。

上の例と同様の処理において、要素の変更前後でその `id` 値を調べる例を次に示す。

例. タプルの要素の変更前後での `id` 値

```
>>> tpl = ('a', [1, 2, 3], 'c') Enter  
>>> id( tpl[1] ) Enter ←リストの要素の id 値を調べる (変更前)  
2158260689472 ←変更前の id 値 (※実際の値は、この例の実行時に決定される)  
>>> tpl[1][1] = 'b' Enter ←リストの内部 (要素) を変更  
>>> tpl Enter ←内容確認  
('a', [1, 'b', 3], 'c') ←リストの部分が変更されている  
>>> id( tpl[1] ) Enter ←リストの要素の id 値を調べる (変更後)  
2158260689472 ←変更後の id 値 (変更前と同じ)
```

この例からわかるように、要素の変更の前後で `tpl[1]` 自体は同一のオブジェクトであり、タプル `tpl` が直接保持する値や参照は変更されていないのでエラーは発生しない。

しかし、タプルの要素 `tpl[1]` に別のオブジェクトを代入しようとするとエラーが発生する。(次の例)

例. タプルの要素 `tpl[1]` に別のオブジェクトを代入する試み (先の例の続き)

```
>>> tpl[1] = [1, 'b', 3] Enter ←タプルの要素の変更を試みると…
Traceback (most recent call last): ←エラーとなる
  File "<stdin>", line 1, in <module>
    tpl[1] = [1, 'b', 3]
    ~~~~~
TypeError: 'tuple' object does not support item assignment
```

この例はタプルの要素 `tpl[1]` 自体の変更を試みるもので、エラーが発生している。

※ イミュータブルなデータ構造でかつ、その要素が参照する全ての内容が完全に不変であるようなオブジェクト (ハッシュ可能なオブジェクト) については、後の「ハッシュ可能なオブジェクト」(p.72) で解説する。

2.5.2.2 括弧の表記が省略できるケース

Python では変数への値の代入をする際に、

```
x, y = 2, 3
```

という並列的な表記が許されている。実はこの例の両辺はタプルになっており、

```
(x, y) = (2, 3)
```

という処理を行ったことと等しい。これはタプルの性質を利用している例であると言える。この例の操作をした直後に `x, y` それぞれの変数の値を確認すると、

```
>>> x Enter
2
>>> y Enter
3
```

と設定されていることが確認できる。このような高度な代入処理に関して後の「2.5.8 データ構造に沿った値の割当て (分割代入)」(p.85) で更に詳しく解説する。

リストに対する文や関数、メソッドの内、データ構造を変更しないものは概ねタプルに対しても使うことができる。

2.5.2.3 特殊なタプル

要素を持たないタプルも存在し、`()` と記述する。また、要素が1つのタプルは要素の後にコンマを付ける必要がある。

例. 空のタプル

```
>>> a = () Enter ←空のタプルを a に設定
>>> a Enter ←内容確認
() ←空のタプルとして保持されていることがわかる
>>> len(a) Enter ←データ長の確認
0 ←結果 (要素の個数は 0)
```

例. 要素数が1のタプル

```
>>> a = (1) Enter ←括弧の中に要素を1つだけ記述する試み
>>> a Enter ←内容確認
1 ←タプルではなく、数値 1
>>> a = (1,) Enter ←要素が1つのタプルの作成
>>> a Enter ←内容確認
(1,) ←タプルとなっている
>>> len(a) Enter ←データ長の確認
1 ←結果 (要素の個数は 1)
```

この例でもわかるように、括弧の中にコンマ無しで要素を1つだけ記述するとタプルとはならないので注意が必要である。

2.5.2.4 タプルの要素を整列 (ソート) する方法

タプルはイミュータブルなデータ構造なので、それ自体を整列することはできないが、`sorted` 関数を使用すると、整列結果をリストの形で取得することができる。

例. タプルの整列

```
>>> a = (5,4,3,2,1) Enter    ←タプルを作成
>>> sorted( a ) Enter    ←それを sorted 関数で整列
[1, 2, 3, 4, 5]          ←整列結果がリストの形で得られている
>>> a Enter    ←元のタプルの内容確認
(5, 4, 3, 2, 1)          ←変化は無い
```

2.5.2.5 タプルか否かの判定

あるオブジェクトの型がタプルか否かを判定するには `type` 関数を用いる。

例. タプルか否かの判定

```
>>> t = (1,2,3) Enter    ←変数 t にタプルを与える
>>> type( t ) Enter    ← type 関数による型の調査
<class 'tuple'>          ←タプルの型
>>> type( t ) is tuple Enter    ← type 関数と is 演算子による型の判定
True                      ←タプルである
```

2.5.3 セット

セットは '{' と '}' で括ったデータ構造であり、集合論で扱う集合に近い性質を持っている。例えば、

```
>>> s = {4,1,3,2,1,3,4,2} Enter
```

としてセットを生成した後に内容を確認すると、

```
>>> s Enter
{1, 2, 3, 4}
```

となっていることがわかる。すなわち、セットでは要素の重複が許されず、要素に順序の概念がない。この例では要素が整列されているように見えるが、セットの要素の順序は一般的には独特な並びとなる。(次の例参照)

例. セットの要素の独特な並び

```
>>> s = {1,4,9,16,25,36,49,64,81,100} Enter    ←  $1^2, 2^2, \dots, 10^2$  のセット
>>> s Enter    ←内容確認
{64, 1, 4, 36, 100, 9, 16, 49, 81, 25}    ←独特な並び
```

この例のようにセットの要素は独特の順序となるが、同じ版の Python 処理系では同じ順序となる。これは、要素の探索(メンバシップ検査)を高速に実行するための Python 処理系内部の仕組み⁷⁶ による。また、上の例は Windows のコマンドプロンプト環境で実行したものであり、別の実行環境によっては出力時にセットの要素の順が整列されることもある。(次の例参照)

例. IPython 環境における実行例

```
In [1]: s = {1,4,9,16,25,36,49,64,81,100} Enter
In [2]: s Enter
Out[2]: {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}    ←セットの要素が整列されて表示されている
```

これは IPython ⁷⁷ の実行環境による例である。

重要)

出力時のセットの要素の順序はスクリプト実行の場合でも異なったものになることがあり、セットの要素の順序が結果に影響を与えるようなプログラミングは避けること。

2.5.3.1 セットの要素となるオブジェクト

セットは完全に不変なオブジェクト(ハッシュ可能なオブジェクト)を要素とする。従ってリストを始めとする可変な(ミュータブルな)オブジェクトはセットの要素に含めることができない。(セットも別のセットの要素にはできない)

⁷⁶ハッシュ表による高速探索を実現している。

⁷⁷Jupyter Notebook の処理環境も IPython を使用している。

例. 様々なオブジェクトをセットの要素に含める試み

```
>>> s = {1,2,3,'4',5,6}  Enter    ←文字列の '4' をセットの要素にする
>>> s  Enter    ←内容確認
{1, '4', 2, 3, 5, 6}    ←セットになっている
>>> s = {1,2,3,(4,5),6}  Enter    ←タプルをセットの要素にする
>>> s  Enter    ←内容確認
{1, 2, 3, 6, (4, 5)}    ←セットになっている
>>> s = {1,2,3,[4,5],6}  Enter    ←リストをセットの要素にしようとする...
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'    ←リストはハッシュ化できない旨のメッセージ
```

■ ハッシュ可能なオブジェクト

ハッシュ可能なオブジェクトとは、イミュータブルであるだけでなく、その要素が参照する要素を全て含めて不変なオブジェクトである。具体的には hash 関数が処理対象にできる（ハッシュ値を算出可能な）オブジェクトである。

例. hash 関数による検査

```
>>> hash( 5 )  Enter    ← int 型は
5                ←ハッシュ可能
>>> hash( 3.14 )  Enter    ← float 型は
322818021289917443    ←ハッシュ可能
>>> hash( 'abc' )  Enter    ← str 型は
7099304858111872342    ←ハッシュ可能
>>> hash( (1,2,3) )  Enter    ←タプルは
529344067295497451    ←ハッシュ可能
```

ハッシュ可能でないオブジェクトを hash 関数に与えるとエラー（TypeError）が発生する。

例. ハッシュ可能でないオブジェクト

```
>>> hash( [1,2,3] )  Enter    ←リストを hash 関数に渡すと
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
    hash( [1,2,3] )
    ~~~~~
TypeError: unhashable type: 'list'
```

各オブジェクトの値に対するハッシュ値は、「同じ内容のオブジェクトであれば、同じプログラム実行中は必ず同じ値になる」ことが保証されているが、異なる内容のオブジェクトであっても、まれにハッシュ値が一致する（衝突する）場合がある。また、str 型や bytes 型など一部の型では、プログラムの実行ごとにハッシュ値が変化する場合がある。

※ hash 関数の戻り値は Python の版や実行環境で異なることがある。

2.5.3.2 セットの生成

基本的には要素の列を '{' と '}' で括ることでセットを記述するが、空セット（空集合）を生成する際は {} という記述をせずに

```
st = set()
```

のように set クラスのコンストラクタを使用する。これは {} という表記が、後で述べる辞書型データの空辞書と区別できない事情があるためであり、このように明に set コンストラクタで生成することになる。また、リストや文字列などのデータ構造を set コンストラクタの引数に与えることもでき、それらデータ構造の要素をセットの要素に変換することができる。

例. リストや文字列を set コンストラクタに与える

```
>>> s = set( [1,2,3] )  Enter    ←リストを set コンストラクタに与える
>>> s  Enter    ←内容確認
{1, 2, 3}    ←セットになっている
>>> s = set( 'abc' )  Enter    ←文字列を set コンストラクタに与える
>>> s  Enter    ←内容確認
{'c', 'a', 'b'}    ←与えた文字列の各文字がセットの要素になっている
```

2.5.3.3 セットに対する各種の操作

● 要素の追加

セットに要素を追加するには add メソッドを使用する。

例. セットへの要素の追加

```
>>> st = set() Enter ←空セットの生成
>>> st.add('a') Enter ←要素'a'の追加
>>> st Enter ←内容確認
{'a'} ←要素が追加されている
```

● 要素の削除

セットの要素を削除するには discard メソッドを使用する。例えばセット st から要素 'a' を削除するには

```
st.discard('a')
```

とする。リストの場合と同様に remove メソッドを使用することもできるが、削除対象の要素がセットの中がない場合にエラー KeyError が発生するので discard メソッドを使用する方が良い。(次の例参照)

例. 要素の削除 (先の例の続き)

```
>>> st.discard('a') Enter ←要素 'a' を削除
>>> st Enter ←セットの内容確認
set() ←空セットになっている
>>> st.discard('a') Enter ←存在しない要素 'a' の削除を試みる
>>> st Enter ←セットの内容確認
set() ←空セット
```

● 要素の取り出しと削除: pop

セットから要素を取り出した後、その要素を当該セットから削除するには pop メソッドを使用する。pop によって取り出される要素の順序は不定である。また、空のセット set() に対して pop を実行すると KeyError となる。(次の例参照)

例. 要素の取り出しと削除

```
>>> s = set([x**2 for x in range(8)]) Enter ←セットの生成
>>> s Enter ←内容確認
{0, 1, 4, 16, 9, 25, 49, 64} ←結果表示
>>> s.pop(), s.pop(), s.pop(), s.pop(), s.pop(), s.pop(), s.pop(), s.pop() Enter ←popの実行
(0, 1, 4, 16, 9, 25, 49, 64) ←処理結果
>>> s Enter ←処理後の内容確認
set() ←空になった
>>> s.pop() Enter ←空セットに対して pop を実行すると…
Traceback (most recent call last): ←エラーとなる.
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

実際には pop によって取り出される要素の順序は、処理対象のセットの要素の順序となる。

● 全要素の削除

セットの全ての要素を削除して空セットにするには clear メソッドを使用する。例えばセット st の全ての要素を削除するには

```
st.clear()
```

とする。既存のセットを空にするには

```
st = set()
```

として空のセットを代入する方法もあるが、この場合は st は別のオブジェクト (識別値の異なるオブジェクト) になる。clear を使用した場合は元のオブジェクトのままである。

● 要素の個数の取得

セットの要素の個数を調べるには `len` 関数を用いる。例えば、セット `st` の要素の個数を得るには `len(st)` とする。

● セットの複製

リストの場合と同様に、`copy` モジュールの `copy` 関数、`deepcopy` 関数でセットの複製が可能であるが、`copy` メソッドで複製ができる。

例. `copy` メソッドによるセットの複製

```
>>> s = {4,3,2,1}  Enter    ←セットの作成
>>> s2 = s.copy()  Enter    ← copy メソッドによる複製
>>> s2  Enter      ←複製の内容確認
{1, 2, 3, 4}        ←元のセット s と同じ
>>> s2.discard(3)  Enter    ←複製のセットの要素を削除
>>> s2  Enter      ←複製の内容確認
{1, 2, 4}           ←要素 3 が削除されている
>>> s  Enter        ←元のセットの内容確認
{1, 2, 3, 4}        ←影響を受けていない
```

● 最小値, 最大値, 合計

`min`, `max`, `sum` 関数でセットの要素の最小値, 最大値, 合計をそれぞれ求めることができる。

例. 最小値, 最大値, 合計

```
>>> s = {4,3,2,1}  Enter    ←セットの作成
>>> min( s )  Enter    ←最小値を求める
1              ←結果
>>> max( s )  Enter    ←最大値を求める
4              ←結果
>>> sum( s )  Enter    ←合計を求める
10             ←結果
```

2.5.3.4 集合論の操作

セットに対する操作と集合論における操作との対応を表 14 に挙げる。

表 14: Python のセットに対する操作と集合論の操作の対応

Python での表記	集合論における操作	戻り値のタイプ	意味
<code>X in S</code>	$X \in S$	真理値	要素 X は集合 S の要素である
<code>X not in S</code>	$X \notin S$	真理値	要素 X は集合 S の要素でない
<code>S1.issubset(S2)</code>	$S1 \subseteq S2$	真理値	集合 $S1$ は集合 $S2$ の部分集合である
<code>S1.issuperset(S2)</code>	$S2 \subseteq S1$	真理値	集合 $S2$ は集合 $S1$ の部分集合である
<code>S1.isdisjoint(S2)</code>	$S1 \cap S2 = \phi$	真理値	集合 $S1$ と集合 $S2$ は共通部分を持たない
<code>S1.intersection(S2)</code>	$S1 \cap S2$	セット	集合 $S1$ と集合 $S2$ の共通集合を生成する
<code>S1.intersection_update(S2)</code>	$S1 \leftarrow S1 \cap S2$	セット	集合 $S1$ と集合 $S2$ の共通集合を $S1$ の内容として更新する
<code>S1.union(S2)</code>	$S1 \cup S2$	セット	集合 $S1$ と集合 $S2$ の和集合を生成する
<code>S1.update(S2)</code>	$S1 \leftarrow S1 \cup S2$	セット	集合 $S1$ と集合 $S2$ の和集合を $S1$ の内容として更新する
<code>S1.difference(S2)</code>	$S1 - S2$	セット	集合 $S1-S2$ (集合の差) を生成する
<code>S1.symmetric_difference(S2)</code>	$S1 \cup S2 - S1 \cap S2$	セット	集合 $S1$ と $S2$ の共通しない要素の集合を生成する

表 14 に示したメソッドと同等の演算子もある。

例. union, intersection, symmetric_difference 各メソッドと同等の演算子

```
>>> {1,2,3} | {2,3,4} Enter ←和集合：union と同等
{1, 2, 3, 4} ←結果
>>> {1,2,3} & {2,3,4} Enter ←共通集合：intersection と同等
{2, 3} ←結果
>>> {1,2,3} ^ {2,3,4} Enter ← symmetric_difference と同等
{1, 4} ←結果
>>> {1,2,3} - {2,3,4} Enter ←集合の差：difference と同等
{1} ←結果
>>> {2,3,4} - {1,2,3} Enter ←集合の差：difference と同等
{4} ←結果
```

この例のように、和集合は「|」、共通集合は「&」、和集合から共通集合を取り除いた集合は「^」、集合の差は「-」で求める方が簡便である。

● 部分集合の判定

表 14 のメソッドを用いて部分集合の判定ができるが、大小比較の演算子 <, <=, >, >= を用いて判定することもできる。

例. 部分集合の判定

```
>>> {2,4,6} <= {1,2,3,4,5,6} Enter ←部分集合の判定
True ←判定結果
>>> {1,2,3} <= {1,2,3} Enter ←部分集合の判定：全要素が同じ場合
True ←判定結果
>>> {1,2,3} < {1,2,3} Enter ←真部分集合の判定
False ←判定結果
```

2.5.3.5 セットか否かの判定

あるオブジェクトの型がセットか否かを判定するには type 関数を用いる。

例. セットか否かの判定

```
>>> s = {1,2,3} Enter ←変数 s にセットを与える
>>> type( s ) Enter ← type 関数による型の調査
<class 'set'> ←セットの型
>>> type( s ) is set Enter ← type 関数と is 演算子による型の判定
True ←セットである
```

2.5.3.6 frozenset

セットとよく似た frozenset というデータ構造もある。frozenset のオブジェクトは生成後に変更ができないが、セット (set) と比較して高速な処理ができる。

例. frozenset の作成

```
>>> fs = frozenset( [1,2,3] ) Enter ← frozenset コンストラクタで生成
>>> fs Enter ←内容確認
frozenset({1, 2, 3}) ←作成された frozenset
```

セットに対する関数やメソッドの内、内容の変更を伴わないものは基本的に frozenset に対しても使用できる。

例. セットの場合と同様の処理（先の例の続き）

```
>>> fs2 = frozenset( [2,3,4] ) Enter ←別の frozenset を作成
>>> fs.intersection(fs2) Enter ←共通集合を求める
frozenset({2, 3}) ←処理結果
>>> fs & fs2 Enter ←&演算子で共通集合を求める
frozenset({2, 3}) ←処理結果
```


2.5.4 辞書型

辞書型オブジェクト⁷⁸ はキーと値のペアを保持するもので、文字通り辞書のような働きをする。辞書型オブジェクトは

```
{キー 1:値 1, キー 2:値 2, ...}
```

と記述する。辞書型オブジェクトの指定したキーに対応する値にアクセスするにはスライスを用いて

辞書型オブジェクト [キー]

と記述する。辞書型オブジェクトの作成と参照の例を次に示す。

例. 辞書型オブジェクトの作成と参照

```
>>> dic = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'} Enter ←辞書を dic に作成
>>> dic['apple'] Enter ←辞書の中のキー 'apple' に対応する値を参照する
'りんご' ←値が得られた
```

辞書型のオブジェクトはキーの値でハッシュ化されており、探索が高速である。

キーと値のペアを新たに辞書に追加するには、

```
dic['banana'] = 'バナナ'
```

などとする。

重要) 辞書型オブジェクトの要素となる「キーと値のペア」を**エントリ**と呼ぶ。キーに使用できるオブジェクトは数値、文字列、タプルといったもの⁷⁹がある。また、リストや辞書オブジェクトといった**ミュータブル**なオブジェクトはキーには使えないことに注意すること。

参考) dict コンストラクタに初期値を与えて辞書型オブジェクトを作成することもできる。

書き方: dict(キー 1=値 1, キー 2=値 2, ...)

この場合の「キー n」には文字列の引用符を付けなくても良い。

例. dict コンストラクタによる辞書型オブジェクトの作成

```
>>> dic = dict( apple='りんご', orange='みかん', lemon='レモン' ) Enter
>>> dic Enter ←内容確認
{'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'}
```

2.5.4.1 空の辞書の作成

空の辞書を生成するには

```
dic = dict()           あるいは       dic = {}
```

とする。また、既存の辞書を空にするには clear メソッドを使用する。

2.5.4.2 エントリの削除

辞書の中の特定のエントリを削除するには del 文が使用できる。例えば、

```
del dic['banana']
```

とすると辞書 dic から 'banana' のキーを持つエントリが削除される。

辞書型オブジェクトに登録されていないキーを参照しようとすると次の例のようにエラー KeyError が発生する。(次の例参照)

例. 存在しないキーへのアクセス (先の例の続き)

```
>>> dic['grape'] Enter ←存在しないキー 'grape' を参照すると…
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>
KeyError: 'grape'
```

また、del 文によって存在しないキーのエントリを削除しようとした場合も同様のエラーとなる。以上のことから、辞

⁷⁸本書内では単に**辞書オブジェクト**と呼ぶこともある。

⁷⁹ハッシュ可能 (hashable) なオブジェクト

書型オブジェクトにアクセスする際は適切に例外処理⁸⁰ をするか、キーが登録されているかを確認する必要がある。あるいは、簡便な方法として、get メソッドで値を取り出す方法がある。(これに関しては後で解説する)

2.5.4.3 エントリの存在検査

辞書にキーが登録されているかを確認するには 'in' を用いる。

例. キーの存在検査 (先の例の続き)

```
>>> 'grape' in dic  Enter  ←辞書 dic にキー 'grape' が存在するか検査
False  ←存在しない。
```

このように、**キー in 辞書型オブジェクト** という式を記述すると、キーが存在すれば True が、存在しなければ False が得られる。

2.5.4.4 get メソッドによる辞書へのアクセス

存在しないキーにアクセスする可能性などを考慮すると、get メソッドで辞書にアクセスするのが安全である。(次の例参照)

例. get メソッドによる辞書へのアクセス (先の例の続き)

```
>>> v = dic.get('lemon')  Enter  ←存在するキー 'lemon' に対する値を v に格納
>>> print( v )  Enter  ←値の確認
レモン  ←値が得られている
>>> v = dic.get('grape')  Enter  ←存在しないキー 'grape' に対する値の取得を試みる
>>> print( v )  Enter  ←値の確認
None  ←値が得られないので None となる
```

存在しないキーに対する戻り値を get メソッドの第 2 引数に指定することができる。

例. 存在しないキーに対する戻り値の設定 (先の例の続き)

```
>>> v = dic.get( 'grape', '???' )  Enter  ←存在しないキーに対する戻り値を '???' に設定
>>> print( v )  Enter  ←値の確認
???  ←値が得られないので '???' となる
```

注) get メソッドは指定したエントリの値を取得するものであり、エントリの値の設定には使用できない。(次の例参照)

例. get メソッドの処理結果に対して値を設定する試み (先の例の続き)

```
>>> dic.get('lemon') = 'れもん'  Enter  ←値を設定しようとする…
File "<stdin>", line 1  ←文法エラーとなる
    dic.get('lemon') = 'れもん'
    ~~~~~
SyntaxError:cannot assign to function call here.Maybe you meant '==' instead of '='?
```

2.5.4.5 全てのエントリの削除

辞書に対して clear メソッドを実行することで、全てのエントリを削除することができる。clear メソッドを実行した後も、対象の辞書は元のオブジェクトのまま (識別値は元のまま) である。

例. clear メソッドによる全エントリの削除 (先の例の続き)

```
>>> id(dic)  Enter  ← dic の識別値の調査
1803804634944  ← dic の識別値
>>> dic.clear()  Enter  ←全エントリの削除
>>> dic  Enter  ←内容確認
{}  ←空になった
>>> id(dic)  Enter  ← dic の識別値は
1803804634944  ←元のまま
```

既存の辞書 dic を空にするには空の辞書を代入する方法もあるが、その方法では dic は別のオブジェクトとなる。

⁸⁰ 「2.5.1.6 例外処理」(p.63) を参照のこと。

例. 空辞書の代入（先の例の続き）

```
>>> dic = dict() [Enter] ←空辞書の代入
>>> dic [Enter] ←内容確認
{} ←空になった
>>> id(dic) [Enter] ← dic の識別値は
1803804797248 ←別のものになった
```

2.5.4.6 辞書に対する pop

辞書に対して pop メソッドを実行（引数にキーを与える）すると値が取り出され、当該エントリが元の辞書から削除される。

例. 辞書に対する pop

```
>>> dic = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'} [Enter] ←辞書の作成
>>> dic.pop('orange') [Enter] ←キー 'orange' に対する値を pop で取り出す
'mikan' ←値が得られた
>>> dic [Enter] ←元の辞書の内容確認
{'apple': 'りんご', 'lemon': 'レモン'} ← pop の処理対象となったエントリが削除されている
```

注) 辞書に対する pop メソッドでは、引数を空にして実行するとエラー（TypeError）が発生する。

2.5.4.7 辞書の更新

辞書の既に存在するキーに対して新たに値を設定すると、その値に更新される。

例. 既存のキーに対する値の再設定

```
>>> dic = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'} [Enter] ←辞書の作成
>>> dic['lemon'] = 'れもん' [Enter] ←既存のキー 'lemon' に新たに値を設定する
>>> dic [Enter] ←内容確認
{'apple': 'りんご', 'orange': 'みかん', 'lemon': 'れもん'} ←値が更新されている
```

これは、1つのエントリの更新の方法である。複数のエントリをまとめて更新することもできる。（次の例参照）

例. 複数のエントリの更新（先の例の続き）

```
>>> dic2 = {'apple': '林檎', 'banana': 'バナナ', 'grape': 'ぶどう'} [Enter] ←新たな辞書を作成
>>> dic.update(dic2) [Enter] ←既存の辞書の内容を新たな辞書の内容で更新
>>> dic [Enter] ←内容確認
{'apple': '林檎', 'orange': 'みかん', 'lemon': 'れもん',
 'banana': 'バナナ', 'grape': 'ぶどう'} ←更新結果
```

この例のように update メソッドを使用して

辞書 1.update(辞書 2)

とすると、辞書 1 に辞書 2 の内容を追加する形で更新を行う。

■ 既存のエントリを優先する形の更新処理

setdefault メソッドを使用すると、既存のエントリを優先する形で辞書を更新することができる。（次の例参照）

例. setdefault による辞書の更新：新規エントリの追加の場合

```
>>> dic = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'} [Enter] ←辞書の作成
>>> dic.setdefault( 'banana', 'バナナ' ) [Enter] ←新エントリ「バナナ」の登録
'バナナ' ←登録結果としてエントリの値 'バナナ' が返される
>>> dic [Enter] ←更新後の内容確認
{'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン', 'banana': 'バナナ'} ←辞書の内容
```

この例では、新しいキー 'banana' を持つエントリが辞書に追加されていることがわかる。次に、既存のキー 'orange' のエントリに対して setdefault で更新を試みる。

例. 既存のキー 'orange' のエントリの更新の試み（先の例の続き）

```
>>> dic.setdefault( 'orange', '蜜柑' )  Enter ←既存の 'orange' のエントリの更新を試みる  
'みかん' ←実行結果として既存のエントリの値 'みかん' が返され…  
>>> dic  Enter ←処理後の辞書の内容を確認すると  
{ 'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン', 'banana': 'バナナ' } ←変更なし
```

setdefault による更新処理では、既存のエントリが優先されることがわかる。

2.5.4.8 辞書の結合

Python3.9 の版から、辞書の和を求める（辞書を結合する）演算子 '|' が使用できる。

書き方： d1 | d2

辞書 d1 と d2 を結合する。d1 と d2 に重複するキーがある場合は d2 の側のエントリが優先される。

例. 辞書の結合

```
>>> d1 = { 'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン' }  Enter ←辞書 d1 の作成  
>>> d2 = { 'lemon': '檸檬', 'banana': 'バナナ', 'grape': 'ぶどう' }  Enter ←辞書 d2 の作成  
>>> d1 | d2  Enter ← d1 と d2 の結合 (d2 を優先) ↓結合結果  
{ 'apple': 'りんご', 'orange': 'みかん', 'lemon': '檸檬', 'banana': 'バナナ', 'grape': 'ぶどう' }  
>>> d2 | d1  Enter ← d2 と d1 の結合 (d1 を優先) ↓結合結果  
{ 'lemon': 'レモン', 'banana': 'バナナ', 'grape': 'ぶどう', 'apple': 'りんご', 'orange': 'みかん' }
```

この処理によって元の辞書は変更されない。

例. 元の辞書の内容確認（先の例の続き）

```
>>> d1  Enter ← d1 の内容確認  
{ 'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン' } ←変化なし  
>>> d2  Enter ← d2 の内容確認  
{ 'lemon': '檸檬', 'banana': 'バナナ', 'grape': 'ぶどう' } ←変化なし
```

参考) Python3.8 以前の版の処理系で辞書を結合するには、後に説明する方法 (keys メソッド) で両方の辞書のキーの並びを取り出してセットに変換し、それらの和集合を求める方法を応用すると良い。

2.5.4.9 辞書を他の辞書の内部に展開する方法

アスタリスク 2 つを辞書の前に記述すると、それを他の辞書の内部に展開することができる。

例. 辞書を他の辞書内に展開する

```
>>> d2 = { 'lemon': '檸檬', 'banana': 'バナナ' }  Enter ←この辞書を,  
>>> { 'apple': 'りんご', 'orange': 'みかん', **d2, 'lemon': 'レモン' }  Enter ←この辞書内に展開する  
{ 'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン', 'banana': 'バナナ' } ←展開結果
```

この例では、'lemon': '檸檬' というエントリが展開された後で新たなエントリとして 'lemon': 'レモン' が与えられているので、結果的に後者のエントリが残る。すなわち、同じキーを持つエントリがある場合は、後から記述されたものが結果として残る。

例. 展開位置を変える試み（先の例の続き）

```
>>> { 'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン', **d2 }  Enter ←展開位置を変更  
{ 'apple': 'りんご', 'orange': 'みかん', 'lemon': '檸檬', 'banana': 'バナナ' } ←展開結果
```

先の例と違うのは 'lemon': 'レモン' のエントリの後に d2 を展開しているので、結果として、'lemon': '檸檬' が残っている。

2.5.4.10 キーや値の列を取り出す方法

辞書型オブジェクトから全てのキーを取り出すには、次のように keys メソッドと list 関数を使用する⁸¹。

⁸¹ 「2.5.6 データ構造の変換」(p.84) に示す方法でも辞書の全キーが得られる。

例. 全てのキーの取り出し

```
>>> dic={'apple':'りんご','orange':'みかん','lemon':'レモン'} Enter ←辞書型オブジェクト生成
>>> k = dic.keys() Enter ←全てのキーの取り出し
>>> k Enter ←内容確認
dict_keys(['apple', 'orange', 'lemon']) ←結果表示
>>> list(k) Enter ←リストに変換する
['apple', 'orange', 'lemon'] ←全キーのリスト
```

このように keys メソッドにより dict_keys(...) というオブジェクト⁸² が生成され、更にそれを list 関数でリストにしている。もちろん

```
list(dic.keys())
```

のようにしても良い⁸³。

同様に、辞書型オブジェクトに対して values メソッド⁸⁴ を使用して、全ての値のリストを生成することもできる。

2.5.4.11 エントリの数を調べる方法

辞書型オブジェクトのエントリの個数を調べるには len 関数を用いる。例えば、辞書型オブジェクト dic のエントリ数を得るには len(dic) とする。

例. 辞書のエントリの数を調べる（先の例の続き）

```
>>> len( dic ) Enter ←辞書の長さの調査
3 ←長さ（エントリ数）が得られている
```

2.5.4.12 リストやタプルから辞書を生成する方法

キーと値のペアを要素として持つリストやタプルを dict()⁸⁵ の引数に与えることで、それらを辞書型オブジェクトに変換することができる。

例. リストから辞書型オブジェクトを生成する

```
>>> dic2 = dict( [['dog','犬'],['cat','猫'],['bird','鳥']] ) Enter ←リストから辞書を生成
>>> dic2 Enter ←内容確認
{'dog':'犬', 'cat':'猫', 'bird':'鳥'} ←結果表示
>>> dic2['cat'] Enter ←辞書オブジェクトとして引用
'猫' ←対応する値が得られている
```

例. リストやタプルから辞書型オブジェクトを生成する

```
>>> dic2 = dict( [('dog','犬'),('cat','猫'),('bird','鳥')] ) Enter ←辞書を生成
>>> dic2 Enter ←内容確認
{'dog':'犬', 'cat':'猫', 'bird':'鳥'} ←結果表示
>>> dic2 = dict( (('dog','犬'),('cat','猫'),('bird','鳥')) ) Enter ←辞書を生成
>>> dic2 Enter ←内容確認
{'dog':'犬', 'cat':'猫', 'bird':'鳥'} ←結果表示
```

参考) 2 文字の文字列を要素とするリストから辞書を生成することができる。

例. 2 文字の文字列をエントリに変換可能

```
>>> a = ['a1','b2','c3'] Enter ←このようなリストを
>>> dict(a) Enter ←辞書に変換する
{'a': '1', 'b': '2', 'c': '3'} ←変換結果
```

ただし、「2 文字の文字列」以外の要素を持つリストは辞書には変換できない。

⁸²ビュー (view) と呼ばれるもので、元の辞書が変更されるとそれに連動してビューも変わる。

⁸³更に簡単に list(dic) としてもキーのリストが得られる。

⁸⁴辞書の値のビューを返す。

⁸⁵辞書のコンストラクタ

例. 辞書に変換できない例

```
>>> a = ['abc','def','ghi'] Enter ←このようなリストは
>>> dict(a) Enter ←辞書に変換
Traceback (most recent call last): ←できない
  File "<stdin>", line 1, in <module> (エラーとなる)
ValueError: dictionary update sequence element #0 has length 3; 2 is required
```

2.5.4.13 辞書の全エントリを列として取り出す方法

辞書オブジェクトに対して items メソッドを実行することで、辞書の全エントリを列の形で取り出すことができる。

例. 辞書の全エントリを列にする（先の例の続き）

```
>>> itm = dic2.items() Enter ←辞書の全内容の取り出し
>>> itm Enter ←内容確認
dict_items([('dog', '犬'), ('cat', '猫'), ('bird', '鳥')]) ←結果表示
>>> list(itm) Enter ←リストに変換
[('dog', '犬'), ('cat', '猫'), ('bird', '鳥')] ←変換結果
```

辞書に対して items メソッドを実行すると dict_items([...]) という形のオブジェクト⁸⁶ が得られる。上の例ではこれを list 関数でリストに変換している。

2.5.4.14 辞書の複製

辞書型オブジェクトに対して copy メソッドを実行することで複製（浅いコピー）することができる。

例. 辞書の複製

```
>>> dic = {'apple':'りんご','orange':'みかん','lemon':'レモン'} Enter ←辞書の作成
>>> dic2 = dic.copy() Enter ←それを複製して dic2 に与える
>>> dic2['orange'] = '蜜柑' Enter ←複製のエントリを変更
>>> dic2 Enter ←複製の内容確認
{'apple':'りんご','orange':'蜜柑','lemon':'レモン'} ←'orange'のエントリが変更されている
>>> dic Enter ←元の辞書の内容確認
{'apple':'りんご','orange':'みかん','lemon':'レモン'} ←元の辞書は変化なし
```

参考) 辞書オブジェクトに対しても copy モジュール⁸⁷ の機能が使用できる。辞書のエントリの深いコピーについても試されたい。

2.5.4.15 エントリの順序

Python3.7 の版から、辞書に登録したエントリの順序が保証された⁸⁸。

例 1. 登録したエントリの順序の確認

```
>>> d1 = {} Enter ←空の辞書 d1 の作成
>>> d1['apple'] = 'りんご' Enter ←エントリの登録（以下同様）
>>> d1['banana'] = 'バナナ' Enter
>>> d1['grape'] = 'ブドウ' Enter
>>> d1 Enter ←内容確認
{'apple': 'りんご', 'banana': 'バナナ', 'grape': 'ブドウ'} ←でき上がった辞書
```

次に、エントリの登録順序を変えて同様の処理を行う例を示す。

⁸⁶ビュー（view）と呼ばれるもので、元の辞書が変更されるとそれに連動してビューも変わる。

⁸⁷「2.5.1.12 リストの複製」（p.66）を参照のこと。

⁸⁸実際には Python3.6 からこの性質がある。

例 2. 登録したエントリの順序の確認（その 2）

```
>>> d2 = {} Enter ←空の辞書 d2 の作成
>>> d2['grape'] = 'ブドウ' Enter ←エントリの登録（以下同様）
>>> d2['banana'] = 'バナナ' Enter
>>> d2['apple'] = 'リンゴ' Enter
>>> d2 Enter ←内容確認
{'grape': 'ブドウ', 'banana': 'バナナ', 'apple': 'リンゴ'} ←でき上がった辞書
```

エントリの順序が登録順になっていることがわかる。

2.5.4.16 辞書の整列

エントリの順序が登録順になるという性質を応用すると、辞書の整列が可能である。例えば、先の例の辞書 d2 をキーの順で整列するには次のような処理を行う。

例. 辞書の整列（先の例の続き）

```
>>> d3 = {} Enter ←空の辞書 d3 の作成
>>> for k in sorted(d2.keys()): Enter ←辞書 d2 のキーを整列したもので反復処理89 する
...     d3[k] = d2[k] Enter ←辞書 d3 に辞書 d2 のエントリを登録する
... Enter ←for 文の記述の終了
>>> d3 Enter ←内容確認
{'apple': 'リンゴ', 'banana': 'バナナ', 'grape': 'ブドウ'} ←整列された辞書
```

この例では、辞書 d2 のキーを `sorted(d2.keys())` で整列し、その順序に従って辞書 d2 のエントリを d3 に登録している。結果として整列された辞書が d3 に得られている。もちろん、元の辞書 d2 には変化はない。（次の例参照）

例. 元の辞書の内容確認（先の例の続き）

```
>>> d2 Enter ←元の辞書 d2 の内容確認
{'grape': 'ブドウ', 'banana': 'バナナ', 'apple': 'リンゴ'} ←辞書 d2 の内容
```

課題. for 文を使わずに辞書を整列する方法について考えよ。

2.5.4.17 辞書の同値性

異なる辞書が同じエントリを持っている場合、`'=='` による同値性の判定は `True` となる。その場合、エントリの順序は関係ない。

例. エントリの順序が異なる同値の辞書の比較

```
>>> d1 = {'dog': '犬', 'cat': '猫'} Enter ←辞書
>>> d2 = {'cat': '猫', 'dog': '犬'} Enter ←上の辞書と同じエントリを持つ辞書
>>> d1 == d2 Enter ←比較結果は
True ←真
```

2.5.4.18 辞書か否かの判定

あるオブジェクトの型が辞書か否かを判定するには `type` 関数を用いる。

例. 辞書か否かの判定

```
>>> d = {'dog': '犬', 'cat': '猫'} Enter ←変数 d に辞書を与える
>>> type(d) Enter ←type 関数による型の調査
<class 'dict'> ←辞書の型
>>> type(d) is dict Enter ←type 関数と is 演算子による型の判定
True ←辞書である
```

⁸⁹詳しくは「2.6 制御構造」(p.89) のところで解説する。

2.5.5 添字（スライス）の高度な応用

リストやタプルあるいは文字列といったデータ構造では、添字 '`...`' を付けることで部分列を取り出すことができるが、ここでは添字（スライス）の少し高度な応用例を紹介する。

2.5.5.1 添字の値の省略

先頭あるいは最終位置の添字は省略することができる。例えば、

```
>>> a = 'abcdefghijklmnopqrstuvwxy' Enter
```

として変数 `a` にアルファベット小文字の列を設定しておくと、`a` の要素は「0 番目」から「25 番目」のインデックスでアクセスできる。このとき `a[0:26]`（0 番目から 26 番目未満）は文字列全体を示す。スライスのインデックス範囲の記述が 0 から始まる場合はそれを省略することができる。同様にインデックスの範囲の記述が「最後の要素のインデックス + 1」で終了する場合もそれを省略できる。（次の例参照）

例. 添字の省略（先の例の続き）

```
>>> a[:26] Enter          ←先頭位置の 0 を省略
'abcdefghijklmnopqrstuvwxy' ←文字列全体
>>> a[0:] Enter          ←最終位置の 26 を省略
'abcdefghijklmnopqrstuvwxy' ←文字列全体
>>> a[:] Enter          ←両方省略
'abcdefghijklmnopqrstuvwxy' ←文字列全体
```

2.5.5.2 スライスオブジェクト

スライスは '`s1 : s2`'、あるいは '`s1 : s2 : s3`' と記述するが、このスライス自体を表現するスライスオブジェクトがあり、これを用いてスライスの部分を

`slice(s1, s2)` あるいは `slice(s1, s2, s3)`

と記述することができる。（次の例参照）

例. スライスオブジェクト

```
>>> lst = [1,2,3,4,5,6,7] Enter ←リストの用意
>>> s = slice(1,4) Enter ←'[1:4]' と同等のスライスオブジェクト
>>> lst[s] Enter          ←内容確認
[2, 3, 4]          ←結果表示
>>> lst[ slice(1,6,2) ] Enter ←'lst[1:6:2]' と同等の記述
[2, 4, 6]          ←結果表示
```

スライスオブジェクトにおいて添字を省略する場合は `None` を指定する。これを用いて先の例と同じ処理を試みる。

例. 添字の省略（その 2）

```
>>> a = 'abcdefghijklmnopqrstuvwxy' Enter ←文字列の用意
>>> s = slice(None,26) Enter ←先頭位置の 0 を省略
>>> a[s] Enter          ←内容確認
'abcdefghijklmnopqrstuvwxy' ←文字列全体
>>> s = slice(0,None) Enter ←最終位置の 26 を省略
>>> a[s] Enter          ←内容確認
'abcdefghijklmnopqrstuvwxy' ←文字列全体
>>> s = slice(None,None) Enter ←両方省略
>>> a[s] Enter          ←内容確認
'abcdefghijklmnopqrstuvwxy' ←文字列全体
```

2.5.5.3 逆順の要素指定

添字に負の数を指定すると、末尾から逆の順に要素を参照することができる。例えば上の例において、`a[-1]` とすると、末尾の要素を参照したことになる。（次の例参照）

例. 逆順の要素指定 (先の例の続き)

```
>>> a[-1] Enter ←末尾の要素を参照
'z' ←末尾の要素
>>> a[-2] Enter ←末尾から 2 番目の要素を参照
'y' ←末尾から 2 番目の要素
```

2.5.5.4 不連続な部分の取り出し

添字は [開始位置:終了位置:増分] のように増分を指定することができ、不連続な「飛び飛びの」部分を取り出すこともできる。(次の例参照)

例. 不連続な位置の要素の取り出し (先の例の続き)

```
>>> a[::2] Enter ←偶数番目の取り出し
'acegikmoqsuw' ←先頭から 1 つ飛びの要素
>>> a[1::2] Enter ←奇数番目の取り出し
'bdfhjlnprtvxz' ←1 番目から 1 つ飛びの要素
```

更に、増分には負 (マイナス) の値を指定することもできる。(次の例参照)

例. 増分に負の値を与える (先の例の続き)

```
>>> a[::-1] Enter ←逆順に取り出す
'zyxwvutsrqponmlkjihgfedcba' ←結果的に反転したことになる
```

参考) スライスオブジェクトを用いた例 (先の例の続き)

```
>>> s = slice(None, None, 2) Enter ←偶数番目の取り出し
>>> a[s] Enter ←内容確認
'acegikmoqsuw' ←先頭から 1 つ飛びの要素
>>> s = slice(1, None, 2) Enter ←奇数番目の取り出し
>>> a[s] Enter ←内容確認
'bdfhjlnprtvxz' ←1 番目から 1 つ飛びの要素
>>> s = slice(None, None, -1) Enter ←逆順に取り出す
>>> a[s] Enter ←内容確認
'zyxwvutsrqponmlkjihgfedcba' ←結果的に反転したことになる
```

2.5.6 データ構造の変換

表 15 のような関数 (コンストラクタ) を使用して、リスト、タプル、セットの間でデータを相互に変換することができる。

表 15: データ構造 d を別の型に変換する関数 (コンストラクタ)

関数 (コンストラクタ)	説 明
list(d)	d をリストに変換する
tuple(d)	d をタプルに変換する
set(d)	d をセットに変換する

例. データ構造の変換

```
>>> list( (1,2,3) ) Enter ←リストへの変換
[1, 2, 3] ←処理結果
>>> tuple( [1,2,3] ) Enter ←タプルへの変換
(1, 2, 3) ←処理結果
>>> set( [1,2,3] ) Enter ←セットへの変換
{1, 2, 3} ←処理結果
```

辞書オブジェクトを表 15 の関数で変換すると、その辞書の全てのキーを要素とするものが得られる。

例. 辞書を他の構造に変換する

```
>>> dic={'apple':'りんご','orange':'みかん','lemon':'レモン'} Enter ←辞書の作成
>>> list(dic) Enter ←リストへの変換
['apple', 'orange', 'lemon'] ←処理結果
>>> tuple(dic) Enter ←タプルへの変換
('apple', 'orange', 'lemon') ←処理結果
>>> set(dic) Enter ←セットへの変換
{'lemon', 'orange', 'apple'} ←処理結果
```

2.5.7 データ構造の要素を他のデータ構造の中に展開する方法

リストやタプル、文字列といったデータ構造の要素を、他のデータ構造の中に展開する方法について説明する。これを応用するとデータ構造の編集が容易になることがある。

例. リストの中に他のリストを展開する

```
>>> x = ['d','e','f'] Enter ←リストを x に用意
>>> ['a','b','c', *x, 'g','h','i'] Enter ←リスト x をこのリストの中に展開
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'] ←得られたリスト
```

このように、展開したいリストの先頭にアスタリスク「*」を付ける。

例. リストの中に複数のリストを展開する

```
>>> x = ['a','b','c'] Enter ←リスト x
>>> y = ['d','e','f'] Enter ←リスト y
>>> z = ['g','h','i'] Enter ←リスト z
>>> [*x, *y, *z] Enter ← x, y, z をこのリストの中に展開
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'] ←得られたリスト
```

文字列の要素を展開することもできる。(次の例)

例. リストの中に文字列を展開する

```
>>> x = 'def' Enter ←文字列を x に用意
>>> ['a','b','c', *x, 'g','h','i'] Enter ←文字列 x をこのリストの中に展開
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'] ←得られたリスト
```

タプルの内部に他のデータ構造を展開することもできる。(次の例)

例. タプルの中に文字列を展開する (先の例の続き)

```
>>> ('a','b','c', *x, 'g','h','i') Enter ←文字列 x をこのタプルの中に展開
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i') ←得られたタプル
```

2.5.8 データ構造に沿った値の割当て (分割代入)

Python では、データ構造に沿った形で変数に値を割り当てること (分割代入: destructuring assignment)⁹⁰ ができる。これにより、データ構造から簡単に部分要素を抽出することができ、データ構造の解析のための簡便なる手段を与える。以下に具体例を示す。

例. データ構造に沿った値の割当て (1)

```
>>> [a,b] = [2,3] Enter ←リストに沿った値の割当て
>>> a Enter ←値の確認
2
>>> b Enter
3 ← a,b 共に値が割当てられていることがわかる
```

⁹⁰ データ構造を分解して部分を取り出すことからアンパック (unpack) と呼ぶこともある。

例. データ構造に沿った値の割当て (2)

```
>>> (a,b) = [3,4] Enter ←タプルに沿った値の割当て (右辺はタプル, リストどちらも可)
>>> a Enter ←値の確認
3
>>> b Enter
4 ← a,b 共に値が割当てられていることがわかる
```

タプルに沿って値を割り当てる際は括弧 ‘()’ を省略することができる. (次の例参照)

例. データ構造に沿った値の割当て (3)

```
>>> a,b = 3,4 Enter ←タプルに沿った値の割当て
>>> a Enter ←値の確認
3
>>> b Enter
4 ← a,b 共に値が割当てられていることがわかる
```

例. データ構造に沿った値の割当て (4)

```
>>> (a,b,c) = (1,2,(3,4)) Enter ←複雑な構造に沿った値の割当て
>>> a Enter ←値の確認
1
>>> b Enter
2
>>> c Enter
(3, 4) ← a,b,c 全て値が割当てられていることがわかる
```

この例の最初の代入文 ‘a,b = 3,4’ は Python では多く用いられる記述形式であるが, これは ‘=’ による代入処理が並列に働いたと見るべきではない. あくまで, データ構造に沿った形で値の割り当て (分割代入) である.

2.5.8.1 応用例 1: 変数の値の交換

分割代入を応用すると, **変数の値の交換**が簡潔に記述できる.

例. 変数の値の交換

```
>>> x = 1 Enter ←変数 x の設定
>>> y = 2 Enter ←変数 y の設定
>>> x,y = y,x Enter ←値の交換
>>> x Enter ←x の値の確認
2
>>> y Enter ←y の値の確認
1
```

2.5.8.2 応用例 2: 高度な代入処理

分割代入の左辺に記述する変数の最後のものにアスタリスク ‘*’ の接頭辞を付けると, 右辺の末尾の複数の要素を対応させることができる.

例. 末尾の複数の要素を分割代入する

```
>>> [a,b,*c] = [1,2,3,4,5,6] Enter ←分割代入
>>> a Enter ←変数 a の値の確認
1 ←変数 a の値
>>> b Enter ←変数 b の値の確認
2 ←変数 b の値
>>> c Enter ←変数 c の値の確認
[3, 4, 5, 6] ←変数 c の値 (末尾の複数の要素が得られている)
```

割り当てるべき変数をアンダースコア ‘_’ で省略することもできる. (次の例参照)

例. 値の割当てにおける変数の省略

```
>>> (a,b,_) = (4,5,6)  Enter  ← ‘_’ による変数の省略
>>> a  Enter  ←値の確認
4
>>> b  Enter
5  ← a,b に値が割当てられていることがわかる
```

ただし厳密には、アンダースコア ‘_’ も変数である。(次の例)

例. アンダースコアに割り当てられたもの(先の例の続き)

```
>>> _  Enter  ←アンダースコアの内容を参照する
6  ←値を持っている
```

2.5.8.3 データ構造の選択的な部分抽出

以上のことを応用すると、複雑なデータ構造から選択的に部分を抽出することができる。(次の例参照)

例. データ構造の部分抽出

```
>>> [a,b,(_,c)] = [1,2,(3,4)]  Enter  ←入れ子になった構造
>>> a,b,c  Enter  ←値の確認
(1, 2, 4)  ←データの部分抽出ができています。
```

2.5.9 データ構造のシャッフル, ランダムサンプリング

random モジュールの shuffle 関数を用いるとリストの要素の順序をシャッフルすることができる。

例. リストの要素のシャッフル

```
>>> import random  Enter  ←モジュールの読み込み
>>> a = [0,1,2,3,4,5,6,7,8,9]  Enter  ←リストを用意
>>> random.shuffle( a )  Enter  ←シャッフルの実行
>>> a  Enter  ←内容確認
[7, 8, 2, 6, 0, 3, 4, 9, 5, 1]  ←シャッフルされている
```

shuffle 関数は、引数に与えたリストオブジェクトそのものを改変する。

データ構造に関する操作と shuffle 関数を組み合わせると、リスト以外のデータ構造もシャッフルすることができる。

例. 文字列のシャッフル(先の例の続き)

```
>>> s = 'abcdefghijklmn'  Enter  ←文字列を用意
>>> t = list( s )  Enter  ←それをリストに変換
>>> t  Enter  ←内容確認
['a','b','c','d','e','f','g','h','i','j','k','l','m','n']  ←リストになった
>>> random.shuffle( t )  Enter  ←それをシャッフルして、
>>> ''.join(t)  Enter  ←全要素を連結すると、
'jbalcigdkfhmne'  ←結果として、文字列をシャッフルしたことになる
```

random モジュールの choice 関数を用いるとデータ構造の要素を1つ無作為抽出(ランダムサンプリング)することができる。

例. 要素を1つ無作為に抽出する(先の例の続き)

```
>>> random.choice( [0,1,2,3,4,5,6,7,8,9] )  Enter  ←引数に与えたリストから無作為抽出
8  ←得られたもの
>>> random.choice( 'abcdefghijkl' )  Enter  ←引数に与えた文字列から無作為抽出
'c'  ←得られたもの
```

複数の要素を重複を許す形で無作為抽出するには choices 関数を用いる。

例. 複数の要素を重複を許して無作為に抽出する（先の例の続き）

```
>>> random.choices( 'abcdefghij', k=4 ) Enter ←引数に与えた文字列から無作為に 4 つ抽出  
['e', 'e', 'b', 'j'] ←得られたもの
```

このように引数「k=抽出個数」を与える。

複数の要素を重複しない形で無作為抽出するには sample 関数を用いる。

例. 複数の要素を重複せず無作為に抽出する（先の例の続き）

```
>>> random.sample( [0,1,2,3,4,5,6,7,8,9], 3 ) Enter ←リストから 3 つを無作為抽出  
[2, 7, 1] ←得られたもの  
>>> random.sample( 'abcdefghij', 3 ) Enter ←文字列から 3 つ無作為抽出  
['h', 'e', 'j'] ←得られたもの
```

2.5.10 データ構造へのアクセスの速度について

Python の基本的なデータ構造であるリスト、セット、辞書それぞれについてアクセスの速度を評価すると概ね次のようになる。

- スライスに整数のインデックスを与えて要素にアクセスする形態（「 n 番目の要素」を指定するアクセス方法）ではリストは高速にアクセスできるデータ構造である。
- in 演算子などで要素のメンバシップを調べる（ある要素が含まれるかを調べる）場合はリストが最も遅く、セットと辞書は共に高速である。（セットと辞書はほぼ同等の速度）

データ構造のアクセス速度の比較を行うサンプルプログラムを付録「I.1 リスト／セット／辞書のアクセス速度の比較」（p.453）で示す。

2.5.11 累算代入によるデータ構造の拡張に関する注意点

文字列、リスト、タプルは累算加算演算子「+=」で拡張することができるが、ミュータブル／イミュータブルの性質に起因する注意点がある。リストはミュータブルなオブジェクトであり、「+=」による拡張の前後で同一のオブジェクトである（識別値が同じに保たれる）が、文字列とタプルはイミュータブルなので「+=」による拡張の前後で異なるオブジェクトとなる。

例. リストの累算加算

```
>>> lst = [1,2,3] Enter ←リストの作成  
>>> id(lst) Enter ←識別値の調査  
1803804792320 ← lst の識別値  
>>> lst += [4,5] Enter ←累算加算によるリストの拡張  
>>> lst Enter ←内容確認  
[1, 2, 3, 4, 5] ← lst が拡張されている  
>>> id(lst) Enter ←識別値の調査  
1803804792320 ←先と同じ識別値（lst は元のオブジェクトのまま）
```

累算代入の前後でリスト lst は同一のオブジェクトであることがわかる。

例. 文字列の累算加算

```
>>> s = 'abc' Enter ←文字列の作成  
>>> id(s) Enter ←識別値の調査  
1710790835920 ← s の識別値  
>>> s += 'def' Enter ←累算加算による文字列の連結  
>>> s Enter ←内容確認  
'abcdef' ← s が拡張されているが、  
>>> id(s) Enter ←その識別値は  
1710796352480 ←元のものとは異なる
```

この例から、文字列は累算加算で別のオブジェクトに変わっていることがわかる。タプルも同様の結果となる。（試されたい）

2.6 制御構造

2.6.1 繰り返し (1): for

処理の繰り返しを実現する文の1つに for がある。

《 for による繰り返し 》

書き方: for 変数 in データ構造:
(変数を参照した処理)

「変数を参照した処理」は繰り返しの対象となるプログラムの部分である。この部分は for の記述開始位置よりも右にインデント (字下げ) する必要がある。またこの部分には複数の行を記述することができるが、その場合は各行に同じ深さのインデントを施さなければならない。

for 文は繰り返しの度に「データ構造」から順番に要素を取り出してそれを「変数」に与える。繰り返しに使用する「データ構造」にはリストの他、タプルや文字列など様々なものが指定できる。

2.6.1.1 イテラブルなオブジェクト

for 文をはじめとする繰り返し制御に使用できるデータ構造のことをイテラブル (iterable) なオブジェクト⁹¹ であるという。先に解説したリスト、タプル、文字列、辞書、セット、後に説明する range オブジェクトはイテラブルなオブジェクトであり、他にもいくつか存在する。特に、セットには順序の概念が無いがイテラブルであることに注意すること。

後の「2.6.1.8 イテレータ」(p.92) で説明するイテレータもイテラブルなオブジェクトであるが、それらは同義ではないことに注意すること。

以後、イテラブルなオブジェクトを単に「イテラブル」と表記することがある。

2.6.1.2 「スイート」の概念

for 文の制御対象となる「同じインデント位置から始まる一連のプログラムの部分」のことをスイート (suite) と呼ぶ。for 文だけでなく、後で解説する各種の制御構造においても「スイート」を制御対象とする。また、関数やクラスの定義 (これらについても後で説明する) においても「スイート」の概念が前提となる。

1つのスイートを構成するためのインデントには、スペースキーで入力する「空白文字」以外にも **Tab** キーで入力する「タブ文字」が使える。ただし、インデントは空白文字やタブ文字の「文字数」で決まるものであるので、インデントの見た目の位置とインデントの文字数には注意が必要 (図5の例) である。

<pre>for i in range(3): print(i,end=',') print(2*i,end=',') print(3*i)</pre>	<p>□□□□ スペース4つ</p> <p>□ タブ文字1つ</p> <p>見た目は同じインデント位置でも、 実際は異なるインデント位置</p>
--	--

図 5: 誤ったインデントの例

for 文を用いたサンプルプログラム test04-1.py を次に示す。

プログラム: test04-1.py

```
1 # coding: utf-8  
2 # 例1: リストの全ての要素を表示する  
3 for word in ['book','orange','man','bird']:  
4     print(word)  
5 # 例2: 繰り返し対象が複数の行の場合  
6 for x in [1,3,5,7]:  
7     print("x=",x)
```

⁹¹ 「反復可能なオブジェクト」という意味。

```

8      print("x^3=",x**3)
9      print("x^10=",x**10)
10     print("x^70=",x**70)
11 # 例3：上記と同様の処理
12 for x in range(1,9,2):          # rangeで作成した数列を与える
13     print("x=",x)
14     print("x^3=",x**3)
15     print("x^10=",x**10)
16     print("x^70=",x**70)

```

このプログラムの例1の部分（3～4行目）は与えられたリストの要素を先頭から1つずつ変数 `word` にセットしてそれを出力するものである。この例1の部分の実行により出力結果は

```

book
orange
man
bird

```

となる。

例2の部分（6～10行目）は1から7までの奇数に対して3乗、10乗、70乗を計算して出力するものである。このように同じインデント（字下げ）を持つ複数の行（スイート）を繰り返しの対象とする⁹² ことができる。この例2の部分の実行により出力結果は

```

x= 1
x ^ 3= 1
x ^ 10= 1
x ^ 70= 1
x= 3
x ^ 3= 27
x ^ 10= 59049
x ^ 70= 2503155504993241601315571986085849
x= 5
x ^ 3= 125
x ^ 10= 9765625
x ^ 70= 8470329472543003390683225006796419620513916015625
x= 7
x ^ 3= 343
x ^ 10= 282475249
x ^ 70= 143503601609868434285603076356671071740077383739246066639249

```

となる。

`for` 文では、与えられたデータ構造の要素を順番に取り出す形で繰り返し処理を実現するが、長大な数列の各項に対して処理を繰り返す場合（例えば1～1億までの繰り返し）には直接にデータ列を書き綴る方法は適切ではない。繰り返し処理に指定する数列の代わりに、後に説明する `range` オブジェクトを使用することができる。先のプログラムの例3の部分（12～16行目）はこれを用いて繰り返しを実現するものであり、例2の部分と同様の動作をする。

2.6.1.3 スイートが1行の場合の書き方

1行のみから成るスイートは `for` と同じ行に記述することができる。

例. 1行のみのスイートを繰り返す `for` 文（対話モードでの例）

```

>>> for i in range(3): print(i)  [Enter]    ← 1行のみのスイートの繰り返し
...  [Enter]                      ← for 文の入力の終了
0                                     ← 出力開始
1
2

```

`for` 文に限らず、他の制御構造、関数やクラスの定義においても、1行のみのスイートはコロン「:」に続けて記述することができる。

2.6.1.4 range 関数, range オブジェクト

`range` クラスのオブジェクトは、実際に要素を書き並べたリストと似た性質を持つ。

⁹²C や Java をはじめとする多くの言語では、ソースプログラムのインデント（字下げ）にはプログラムとしての意味はない。制御対象のプログラムの範囲をインデントによって指定する言語は Python を含め少数派である。

《 range オブジェクト 》

書き方 1:	<code>range(<i>n</i>)</code>	意味:	0 以上 <i>n</i> 未満の整数列
書き方 2:	<code>range(<i>n</i>₁,<i>n</i>₂)</code>	意味:	<i>n</i> ₁ 以上 <i>n</i> ₂ 未満の整数列
書き方 3:	<code>range(<i>n</i>₁,<i>n</i>₂,<i>n</i>₃)</code>	意味:	<i>n</i> ₁ 以上 <i>n</i> ₂ 未満の範囲の公差 <i>n</i> ₃ の整数列

書き方 1 の例. `range(10)` → `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` と同等

書き方 2 の例. `range(5,10)` → `[5, 6, 7, 8, 9]` と同等

書き方 3 の例. `range(1,10,2)` → `[1, 3, 5, 7, 9]` と同等

この range オブジェクトを応用したのが先のプログラム test04-1.py の例 3 (12~16 行目) の部分である。

range オブジェクトは、基本的にはリストと同様の扱いが可能で、長大な数列を扱う場合に記憶資源管理の面で有利である。range オブジェクトを生成する際の `range(…)` の記述は range 関数⁹³ である。

例. range オブジェクトの扱い

```
>>> r = range(10)  [Enter]    ← range オブジェクトの生成
>>> r               [Enter]    ← 内容の確認
range(0, 10)        [Enter]    ← 結果の表示
>>> type( r )       [Enter]    ← データ型の確認
<class 'range'>     ← 'range' 型であることがわかる
>>> r[3]             [Enter]    ← 要素 (3 番目) を取り出すことができる。
3                   [Enter]    ← 結果の表示
>>> list( r )        [Enter]    ← リストに変換
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  ← 変換結果
```

2.6.1.5 辞書の for 文への応用

辞書型オブジェクトを for 文に与えると、辞書のキーが順番に得られる。

例. for 文に辞書を与える

```
>>> d = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン'} [Enter] ← 辞書型オブジェクト
>>> for e in d: [Enter]    ← 辞書オブジェクトのキーが e に順番に得られる
...     print( e, '->', d[e] ) [Enter]    ← 辞書オブジェクトのキーと値を表示
... [Enter]    ← for 文の終了
apple -> りんご    ← 以降、出力
orange -> みかん
lemon -> レモン
```

2.6.1.6 for 文における else

for 文の記述に else を付けると、終了処理を実現できる。

《 else を用いた for 文の終了処理 》

書き方: `for 変数 in データ構造:`
 (変数を参照した処理)
 `else:`
 (終了処理)

for による繰り返しが終了した直後に 1 度だけ「終了処理」を実行する。ただし、break (p.97「2.6.3 繰り返しの中断とスキップ」で解説する) で繰り返しを中断した場合は「終了処理」は実行されない。

これは Python 以外の言語ではあまり見られない便利な構文である。

⁹³厳密には range クラスのコンストラクタである。

2.6.1.7 for を使ったデータ構造の生成（要素の内包表記）

for の別の使い方として、データ構造の生成がある。次の例について考える。

例. 整数の 2 乗のリストを作る

```
>>> [ x**2 for x in range(10) ]  Enter    ←リストの生成
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]    ←得られたリスト
```

このようにして、様々な値のリストを生成できる。

《 for を使ったデータ構造の生成 》：要素の内包表記

リストの生成： [繰り返し変数を使った要素の表現 for 繰り返し変数 in データ構造]

集合、辞書の生成： { 繰り返し変数を使った要素の表現 for 繰り返し変数 in データ構造 }

例. 文字列中の文字を要素とするリスト⁹⁴

```
>>> [c for c in 'abcde']  Enter    ←リストの生成
['a', 'b', 'c', 'd', 'e']    ←得られたリスト
```

このように、in の後ろにイテラブルなデータ構造を与えてリストを生成できる。

例. 辞書の生成

```
>>> d = { x:x**2 for x in range(10) }  Enter    ← 0～9 までの 2 乗の値を保持する辞書 d の生成
>>> d  Enter    ←内容の確認
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> d[4]  Enter    ←キー 4 に対する値                ↑結果の表示
16          ←結果の表示
```

例. セットの生成

```
>>> { x**2 for x in range(10) }  Enter    ← 0～9 までの 2 乗の値を保持するセットの生成
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}    ←結果の表示
```

▲注意▲ 要素の内包表記によるタプルの生成では、ジェネレータという特別なデータ列となる。

例. 内包表記でタプルを作る試み

```
>>> t = ( 2*x for x in range(4) )  Enter    ←内包表記の形式によるタプルの生成
>>> t    ←内容確認
<generator object <genexpr> at 0x0000013C97AFC5C8> ←「ジェネレータ」となっている
>>> list( t )  Enter    ←リストに変換可能
[0, 2, 4, 6]    ←結果の表示
```

ジェネレータは制御のための高度な方法を与える。詳しくは「4.7 ジェネレータ」(p.297) で解説する。

■ 条件付きの内包表記

要素の内包表記において if を記述して条件を付けることができる。

例. 0～19 の範囲の 3 の倍数を求める

```
>>> [ x for x in range(20) if x%3==0 ]  Enter    ← if で条件を付ける
[0, 3, 6, 9, 12, 15, 18]    ←結果の表示
```

if の後に記述する条件式に関しては「2.6.4.1 条件式」(p.98) で解説する。

2.6.1.8 イテレータ

順番に値を取り出すデータ構造にイテレータがある。イテレータは要素を取り出す度にその要素が削除されてゆくデータ構造であり、for などの繰り返し処理のためのイテラブルなオブジェクトとして用いることができる。イテレータは関数 iter で生成することができる。

⁹⁴この例はデータ構造の分解を簡単な形で示すためのものである。同様の処理はより簡単に list('abcde') として実現できる。

例. イテレータによる繰り返し処理

```
>>> it = iter( [1,2,3] )  [Enter]    ← リスト [1,2,3] をイテレータに変換
>>> for m in it:  [Enter]    ←繰り返し処理の記述開始
...     print(m)  [Enter]    ←要素を表示する処理
...  [Enter]    ←繰り返し処理の記述終了
1
2
3                ←出力（ここから）
                ←出力（ここまで）
```

この例を見る限り、it はリスト [1,2,3] と同様のオブジェクトのように思われるが、次の例について考える。

例. next 関数による要素の取り出し

```
>>> it = iter( [1,2,3] )  [Enter]    ← リスト [1,2,3] をイテレータに変換
>>> next(it)  [Enter]    ←次の要素の取り出し
1            ←最初の要素
>>> next(it)  [Enter]    ←次の要素の取り出し
2            ←次の要素
>>> next(it)  [Enter]    ←次の要素の取り出し
3            ←最後の要素
>>> next(it)  [Enter]    ←次の要素の取り出しを試みると…
Traceback (most recent call last):    ←要素が無いことによる例外が発生
  File "<stdin>", line 1, in <module>
StopIteration
```

この例の様に、関数 next によってイテレータの次の要素が取り出される。ただし、取り出すべき次の要素が無い場合は例外 StopIteration が発生する。

イテレータは、リストのような要素を保持するためのデータ構造として扱うべきではなく、「繰り返し構造」の処理対象として扱うべき特殊なデータ構造である。

ここでは説明を簡単にするため、リストからイテレータを生成する処理を例示した。実際には、ストリーム（入力装置や通信装置をはじめとするデータの発生源）を含んだ多くのものがイテレータのような振る舞いをする⁹⁵。

イテレータは要素を参照することでその要素が取り除かれるものであり、その意味では破壊的なデータ構造である。従って、イテレータの内容を破壊されないものとして扱うには、リストなどの別のデータ構造に変換する必要がある。例えば、イテレータ it を次のようにしてリスト L に変換することができる。

```
L = list( it )
```

ただし、この処理の後、it の内容は失われる。（it の参照が終了したことによる）また、得られる要素数が長大になる可能性のあるイテレータ（ファイル入力など）や、終端時期が不明のイテレータ（通信デバイスからの入力など）を list コンストラクタなどで一括変換すべきではない。

イテレータはイテラブルなオブジェクトであるが、それらは同義ではないことに注意すること。（次の例）

例. リストがイテレータでないことの確認

```
>>> lst = [1,2,3]  [Enter]    ←リスト lst の作成
>>> next( lst )  [Enter]    ←それに対して next 関数の実行を試みると…
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
```

next 関数はイテレータ（とその拡張クラス）に対してのみ実行できるものであり、この例ではリストに対して実行できないことが示されている。すなわちイテレータとは、next 関数を始めとする特定の仕様を満たすものとして定義される。プログラマが独自のイテレータを定義する方法に関しては、後の「2.9.7 イテレータのクラスを実装する方法」（p.169）で解説する。

イテレータと関連の深いものにジェネレータがある。これについては「4.7 ジェネレータ」（p.297）で説明する。

⁹⁵厳密には、ファイルや通信装置はイテレータではない。

2.6.1.9 分割代入を用いた for 文

for の後に変数のタプルを与えることで、データ構造の分割代入ができる。

例. 分割代入でデータ構造の要素を分解して取り出す

```
>>> q = [[1,2],[3,4],[5,6]] Enter ←リストを要素とするリスト q
>>> for (x,y) in q: Enter ← q の要素を x, y に分解しながら取り出す
...     print('x=',x,'y=',y) Enter ←それらを出力
... Enter ← for 文の終了
x= 1 y= 2 ←出力
x= 3 y= 4
x= 5 y= 6
```

この方法は、後に説明する zip オブジェクトや enumerate を用いた繰り返し処理において役立つ。

for の後のタプルは括弧 '(...)' を省略して 'for x,y in q:' などと記述しても良い。

2.6.1.10 zip 関数と zip オブジェクト

zip 関数を用いると、複数のイテラブルオブジェクト（リストなど）を束ねて1つのイテラブル（zip オブジェクト）を生成することができる。（次の例参照）

例. 3つのリストを束ねて1つのイテラブルを生成する (1)

```
>>> q1 = [1,2,3] Enter ← 1 番目のリストの生成
>>> q2 = ['one','two','three'] Enter ← 2 番目のリストの生成
>>> q3 = ['一','二','三'] Enter ← 3 番目のリストの生成
>>> z = zip( q1, q2, q3 ) Enter ← それら 3 つを束ねて z にする
>>> for (a,b,c) in z: Enter ← z から要素を 1 つずつ取り出すループ
...     print(a,':',b,':',c) Enter ← 出力処理
... Enter ← 繰り返し記述範囲の終わり
1 : one : 一 ← 出力 (1 番目)
2 : two : 二 ← 出力 (2 番目)
3 : three : 三 ← 出力 (3 番目)
```

これは、3つのリスト q1, q2, q3 を束ねて1つのイテラブル z を生成し、それを用いて繰り返し処理を行っている例である。

zip 関数で得られるオブジェクトは「zip オブジェクト」と呼ばれる特殊なイテラブルである。（次の例参照）

例. 3つのリストを束ねて1つのイテラブルを生成する (2)

```
>>> q1 = [1,2,3] Enter ← 1 番目のリストの生成
>>> q2 = ['one','two','three'] Enter ← 2 番目のリストの生成
>>> q3 = ['一','二','三'] Enter ← 3 番目のリストの生成
>>> z = zip( q1, q2, q3 ) Enter ← それら 3 つを束ねて z にする
>>> z Enter ← z を確認
<zip object at 0x000002E1BC3B9D08> ← zip オブジェクトであることがわかる
>>> list(z) Enter ← z をリストに変換して内容を確認
[(1, 'one', '一'), (2, 'two', '二'), (3, 'three', '三')] ← 内容表示
>>> list(z) Enter ← z の内容を再度確認すると...
[] ← 参照後なので空になっている
```

長さの異なるデータ列を zip で束ねると、最も長さの短い列にイテラブルのサイズが制限される。

例. 長さの異なるリストを束ねて zip オブジェクトを生成する

```
>>> q1 = [1,2,3,4]      Enter      ←長いリスト
>>> q2 = ['one','two','three'] Enter   ←短いリスト
>>> z = zip( q1, q2 )    Enter   ←それらを束ねて z にする
>>> for (a,b) in z:      Enter   ← z から要素を 1 つずつ取り出すループ
...     print(a,':',b)  Enter   ←出力処理
...     Enter           ←繰り返し記述範囲の終わり
1 :   one               ←出力 (1 番目)
2 :   two               ←出力 (2 番目)
3 :   three             ←出力 (3 番目: 短い方のリストのサイズ)
```

【zip オブジェクトの展開】

zip 関数によって束ねられた zip オブジェクトはアスタリスク '*' を用いて他のデータ構造 (リストなど) の内部に展開することができる。

例. リストの内部に zip オブジェクトを展開する

```
>>> a = ['a1','a2','a3'] Enter   ← 1 番目のリストの生成
>>> b = ['b1','b2','b3'] Enter   ← 2 番目のリストの生成
>>> z = zip(a,b)      Enter   ← zip 関数で束ねる
>>> ['a0','b0',*z,'a4','b4'] Enter   ←それをリストの内部に展開する
['a0','b0',('a1','b1'),('a2','b2'),('a3','b3'),'a4','b4'] ←展開結果
```

これは「2.5.1.14 リストの内部に他のリストを展開する方法」(p.68) で解説した内容と同じ手法である。また同様の方法で、関数呼び出しの際の引数並びに zip オブジェクトを展開して渡すことができる。(次の例参照)

例. zip オブジェクトの展開

```
>>> a = ['a1','a2','a3'] Enter   ← 1 番目のリストの生成
>>> b = ['b1','b2','b3'] Enter   ← 2 番目のリストの生成
>>> c = ['c1','c2','c3'] Enter   ← 3 番目のリストの生成
>>> z = zip(a,b,c)      Enter   ← zip 関数で束ねる
>>> print( *z )        Enter   ← zip オブジェクトを展開して print 関数に渡す
('a1','b1','c1') ('a2','b2','c2') ('a3','b3','c3') ← zip オブジェクトの要素が順番に表示される
```

ここで注意しなければならない点がある。zip オブジェクトをアスタリスク '*' で展開したものを単独で取得することはできない。(次の例参照)

例. zip オブジェクトの展開 (失敗例: 先の続き)

```
>>> z = zip(a,b,c)      Enter   ← zip 関数で束ねる
>>> *z                  Enter   ← zip オブジェクトの展開
File "<stdin>", line 1      ←エラー発生
SyntaxError: can't use starred expression here    ←文法エラー
```

関数の引数に '*' を記述することに関しては、後の「2.8.1.3 引数の個数が不定の関数」(p.139) で詳しく説明する。

■ 参考

関数呼び出し時のアスタリスク '*' で展開された要素は、再び zip 関数で束ねることができる。このことが理解できる例を次に示す。

例. アスタリスク '*' による展開と再 zip 化の例 (先の続き)

```
>>> z = zip(a,b,c)      Enter   ← zip 関数で束ねる
>>> print( *zip(*z) )    Enter   ←展開, zip, 展開と連鎖実行
('a1','a2','a3') ('b1','b2','b3') ('c1','c2','c3') ←結果
```

このような結果となる理由に関して考察されたい。

2.6.1.11 enumerate によるインデックス情報の付与

イテラブルから要素を取り出しながら繰り返し処理を行う際、処理対象の要素のインデックスが処理に求められることがしばしばある。(下記の例)

例. データ要素のインデックスを繰り返し処理の中で使用する例

```
>>> s = 'あいう' Enter ←処理対象のデータ列
>>> i = 0 Enter ←要素のインデックスの初期化
>>> for m in s: Enter ←繰り返し処理の記述開始
...     print(i, ' 番目は「',m,'」') Enter ←要素のインデックスを用いた処理
...     i += 1 Enter ←次のインデックスを算出
... Enter ←繰り返し処理の記述終了
0 番目は「 あ 」 ←処理結果
1 番目は「 い 」 ←処理結果
2 番目は「 う 」 ←処理結果
```

この例では繰り返し処理の前に変数 *i* を用意して開始のインデックス 0 を設定し、繰り返し処理の度に 1 を加えるという手法で、処理中の要素のインデックスを得ている。enumerate を用いると、同様の処理を更に簡潔な形で実現することができる。

例. enumerate を用いた実装

```
>>> s = 'あいう' Enter ←処理対象のデータ列
>>> for (i,m) in enumerate(s): Enter ←繰り返し処理の記述開始（インデックス情報付き）
...     print(i, ' 番目は「',m,'」') Enter ←要素のインデックスを用いた処理
... Enter ←繰り返し処理の記述終了
0 番目は「 あ 」 ←処理結果
1 番目は「 い 」 ←処理結果
2 番目は「 う 」 ←処理結果
```

enumerate で生成されたオブジェクトは **enumerate オブジェクト** であり、イテラブルである。(次の例参照)

例. enumerate オブジェクトの内容確認 (先の例の続き)

```
>>> enumerate(s) Enter ←内容確認
<enumerate object at 0x0000015A80298E58> ← enumerate オブジェクトであることがわかる
>>> list( enumerate(s) ) Enter ←リストに変換して内容確認
[(0, 'あ'), (1, 'い'), (2, 'う')] ←各要素がインデックス付きのタプルとなっている
```

enumerate による付番の開始値を変更するには、enumerate の 2 番目の引数に開始値を与える。

例. enumerate による付番の開始値を 1 にする (先の例の続き)

```
>>> list( enumerate(s,1) ) Enter ← enumerate の第 2 引数に開始値を与える
[(1, 'あ'), (2, 'い'), (3, 'う')] ← 1 からの付番となっている
```

付番の開始値はキーワード引数 **start=開始値** として与えることもできる。

例. enumerate による付番の開始値を 2 にする (先の例の続き)

```
>>> list( enumerate(s,start=2) ) Enter ←キーワード引数 start に開始値を与える
[(2, 'あ'), (3, 'い'), (4, 'う')] ← 2 からの付番となっている
```

2.6.2 繰り返し (2): while

条件判定⁹⁶ に基いて処理を繰り返すための while 文がある。

⁹⁶条件の記述に関しては「2.6.4.1 条件式」(p.98) にまとめている。

《 while による繰り返し 》

書き方： while 条件:
 (繰り返し対象の処理)

「条件」を満たす間「繰り返し対象の処理」を繰り返す。「繰り返し対象の処理」(スイート)は while の記述開始位置よりも右にインデント (字下げ) される必要がある。(for 文の場合と同様)

《 else を用いた while 文の終了処理 》

書き方： while 条件:
 (繰り返し対象の処理)
 else:
 (条件が不成立の場合の処理)

「条件」が不成立になった場合に 1 度だけ「条件が不成立の場合の処理」を実行して while 文を終了する。ただし、後で説明する break で繰り返しを中断した場合は「条件が不成立の場合の処理」の部分は実行されない。

while 文を用いたサンプルプログラム test04-2.py を次に示す。

プログラム：test04-2.py

```
1 # coding: utf-8
2 n = 0
3 while n < 10:
4     print(n)
5     n += 2
6 else:
7     print('end')
```

このプログラムの実行により出力結果は

```
0
2
4
6
8
end
```

となる。

このサンプルの 5 行目の '+' は累算代入演算子⁹⁷である。すなわちこれは、

```
n = n + 2
```

と記述したものと同等である。

2.6.3 繰り返しの中断とスキップ

for や while による処理の繰り返しは break で中断して抜け出すことができる。また、繰り返し対象の部分で continue を使うと、繰り返し処理を次の回にスキップできる。(C や Java と同じ)

2.6.4 条件分岐

条件判定により実行する処理を選択するには if 文を使用⁹⁸する。

《 if 文による条件分岐 》 (その 1)

書き方： if 条件:
 (対象の処理)

「条件」が成立したときに「対象の処理」を実行する。「対象の処理」(スイート)は if の記述開始位置よりも右にインデント (字下げ) される必要がある。(for 文の場合と同様)

⁹⁷C や Java のそれと同じ働きを持つ。

⁹⁸Python には C や Java のような switch 文はないが、Python3.10 から高度な分岐処理のための構文が導入された。これに関しては「2.12 構造的パターンマッチング」(p.192)で解説する。

《 if 文による条件分岐 》（その 2）

書き方： if 条件:

（対象の処理）

else:

（条件が不成立の場合の処理）

「条件」が不成立であった場合に else 以下の「条件が不成立の場合の処理」を実行する。

《 if 文による条件分岐 》（その 3）

書き方： if 条件 1:

（条件 1 を満たした場合の処理）

elif 条件 2:

（条件 2 を満たした場合の処理）

:

else:

（全ての条件が不成立の場合の処理）

複数の条件分岐を実現する場合にこのように記述する。

2.6.4.1 条件式

条件として記述できるものは表 16 のような**比較演算子**を用いた式や、それらを**論理演算子**（表 17）で結合（装飾）した式である。

表 16: 比較演算子

比較演算子を用いた条件式	説 明
<code>a == b</code>	a と b の値が等しい場合に True, それ以外の場合は False.
<code>a != b</code>	a と b が異なる場合に True, 等しい場合は False.
<code>a > b</code>	a が b より大きい場合に True, それ以外の場合は False.
<code>a >= b</code>	a が b 以上の場合に True, それ以外の場合は False.
<code>a < b</code>	a が b より小さい場合に True, それ以外の場合は False.
<code>a <= b</code>	a が b 以下の場合に True, それ以外の場合は False.

表 17: 論理演算子

論理演算子を用いた条件式	説 明
<code>P and Q</code>	P と Q が共に True の場合に True, それ以外の場合は False.
<code>P or Q</code>	P と Q の少なくとも 1 つが True の場合に True, それ以外の場合は False.
<code>not P</code>	P が False の場合に True, それ以外の場合は False.

条件式はそれ自体が真理値の値（True か False）を返す。

※ 「if～else…」の構文は 3 項演算子として記述することも可能である。これに関しては「2.10.4 3 項演算子としての if～else…」(p.188) のところで解説する。

2.6.4.2 比較演算子の連鎖

表 16 の比較演算子は連鎖する形で記述することができる。

例. 比較演算子の連鎖：その 1

```
>>> x = 3 
```

```
>>> 1 < x < 4  ←比較演算子の連鎖
```

```
True ←判定結果
```

更に長く記述することもできる。

例. 比較演算子の連鎖：その2（先の例の続き）

```
>>> y,z = 5,7 
>>> 1 < x < y < z < 9  ←比較演算子の連鎖
True      ←判定結果
>>> 9 > z > y > x > 1  ←比較演算子の連鎖
True      ←判定結果
```

‘==’, ‘!=’ も使用できる.

例. 比較演算子の連鎖：その3（先の例の続き）

```
>>> 3 == x < y  ←‘==’を含む判定
True      ←判定結果
>>> 1 != x < y  ←‘!=’を含む判定
True      ←判定結果
```

比較演算子を連鎖することで複雑な比較を簡潔に記述することができる.

2.6.4.3 データ構造の比較

文字列, リスト, タプルといったデータ構造は, 要素とその並び方により順序が決定され, 順序に基づいて >, >=, <, <= の比較演算が実行される. 例えば, 文字列の場合は構成要素の文字の順序 (文字コードの値に基づく大小判定) で文字列全体の大小関係が決まり, これは英和辞書, 国語辞書などの語順を参考にすると理解しやすい. リストやタプルにおいても, 要素とその並び方により大小関係が決定される.

例. 文字列の比較

```
>>> 'apple' < 'orange' 
True
>>> 'apple' > 'orange' 
False
```

例. リストの比較 (1)

```
>>> [1,2,3] < [1,2,3,4] 
True
>>> [1,2,3] > [1,2,3,4] 
False
```

例. リストの比較 (2)

```
>>> [7,8] > [1,2,3] 
True
>>> [7,8] < [1,2,3] 
False
```

異なる型のデータ構造同士を >, >=, <, <= で判定しようとするとエラー (TypeError) となるので注意すること. また, リストやタプル同士をそれら演算子で比較する場合も, 相互で対応する要素の型が異なると同じエラーが起こることにも注意しなければならない.

例. 異なる型のデータ構造の比較

```
>>> 'apple' < ['o','r','a','n','g','e']  ←文字列とリストの比較
Traceback (most recent call last):      ←エラーとなる
  File "<python-input-36>", line 1, in <module>
    'apple' < ['o','r','a','n','g','e']
TypeError:  '<' not supported between instances of 'str' and 'list'
```

例. 対応する要素の型が異なる場合のリストの比較

```
>>> [1,2] < ['3',4]  ←このような比較は
Traceback (most recent call last):      ←エラーとなる
  File "<python-input-39>", line 1, in <module>
    [1,2] < ['3',4]      ←互いのリストの先頭要素の型が異なることがエラーの原因
TypeError:  '<' not supported between instances of 'int' and 'str'
```


2.6.4.4 各種の「空」値に関する条件判定

if 文の条件式の部分に各種の「空」値を与えた場合の条件判定がどのようになるかをサンプルプログラム emptyCheck0.py の実行によって例示する。このプログラムは、変数 cnd に各種の「空」値を与え、それを条件式として if 文で判定するものである。

プログラム：emptyCheck0.py

```
1  # coding: utf-8
2  # 各種の '空' 値に関する判定
3
4  #--- Noneを条件式に与えた場合 ---
5  cnd = None
6  if cnd:
7      print('hit')
8  else:
9      print(cnd,'は偽の扱いです. \n')
10
11 #--- Noneであるかどうかの判定 ---
12 if cnd is None:
13     print(cnd,'is None による判定')
14     print(cnd,'は None です. \n')
15
16 if cnd is not None:
17     print('hit')
18 else:
19     print(cnd,'is not None による判定')
20     print(cnd,'は None です. \n')
21
22 #--- 空タプル '()' を条件式に与えた場合 ---
23 cnd = ()
24 if cnd:
25     print('hit')
26 else:
27     print(cnd,'は偽の扱いです. ')
28
29 #--- 空リスト '[]' を条件式に与えた場合 ---
30 cnd = []
31 if cnd:
32     print('hit')
33 else:
34     print(cnd,'は偽の扱いです. ')
35
36 #--- 空集合 'set()' を条件式に与えた場合 ---
37 cnd = set()
38 if cnd:
39     print('hit')
40 else:
41     print(cnd,'（空集合）は偽の扱いです. ')
42
43 #--- 空辞書 '{}' を条件式に与えた場合 ---
44 cnd = dict()
45 if cnd:
46     print('hit')
47 else:
48     print(cnd,'（空辞書）は偽の扱いです. ')
49
50 #--- ゼロ 0 を条件式に与えた場合 ---
51 cnd = 0
52 if cnd:
53     print('hit')
54 else:
55     print(cnd,'は偽の扱いです. ')
56
57 #--- 空文字列 '' を条件式に与えた場合 ---
58 cnd = ''
59 if cnd:
60     print('hit')
61 else:
62     print('空文字列',cnd,'は偽の扱いです. ')
```

このプログラムを実行した結果の例を次に示す。

```
None は偽の扱いです.  
  
None is None による判定  
None は None です.  
  
None is not None による判定  
None は None です.  
  
() は偽の扱いです.  
[] は偽の扱いです.  
set() (空集合) は偽の扱いです.  
{ } (空辞書) は偽の扱いです.  
0 は偽の扱いです.  
空文字列 '' は偽の扱いです.
```

このプログラムの実行によって、次に示す値が条件式として「偽」となることがわかる。

None (ヌルオブジェクト)	() (空のタプル)	[] (空リスト)	set() (空セット)
{ } (空の辞書)	0 (ゼロ)	'' (空文字列)	

「空」値でないもの（非「空」値）を条件式に与えると基本的には「真」となる。また、ヌルオブジェクトかどうかを判定するには `is None` や `is not None` を用いて記述する。（`is` に関しては後の「2.6.4.5 `is` 演算子による比較」を参照のこと）

▲注意▲ 「空」値と非「空」の論理演算

「空」値と非「空」を組合せた論理演算（`and`, `or`, `not`）には注意すること。すなわち、「空」値を「偽」、非「空」値を「真」と見做して論理和や論理積といった演算をすると、結果としてどのような値となるかは予め個別に確認した方がよい。

真理値以外の値を真理値として解釈した結果を調べるには `bool` コンストラクタを使用すると良い。

例. `bool` コンストラクタによる確認

```
>>> bool( 0.0 )  Enter    ←浮動小数点数 0.0 を真理値として解釈すると  
False           ←偽となる
```

■ 多量のデータに対する条件の一括判定

「2.10.5 `all`, `any` による一括判定」(p.188) で解説する `all` 関数や `any` 関数を応用すると、多量のデータに対する条件判定を一括して実行することが可能となる。

2.6.4.5 `is` 演算子による比較

「同じ値か」どうかを判定するには演算子 `==` を用いるが、これに対して同一のオブジェクトかどうかを判定する場合には `'is'` を用いる。

次の例のような、2つの変数に割り当てられたリストの比較について考える。

例. リストの比較

```
>>> a = [1,2,3]  Enter    ←変数 a に与えられたリスト  
>>> b = [1,2,3]  Enter    ←変数 b に与えられたリスト  
>>> a == b  Enter    ←変数 a,b の値が同じかどうかを検査  
True       ←真 (a,b の値は同じである)
```

次に `'is'` を用いて検査する。

例. リストの比較 (先の例の続き)

```
>>> a is b  Enter    ←変数 a,b が同一のオブジェクトかどうかを検査  
False      ←偽 (a,b は別のオブジェクトである)
```

この例から「a,b はそれぞれ異なるオブジェクトである」ことがわかる。これは「変数 a,b はそれぞれ、同じ値を持つ別々のオブジェクトである」と言い換えることができる。

オブジェクトの比較においては「同値」であることと**同一性**の違いを意識するべきである。

Python 処理系の中で扱われるオブジェクトは、それが廃棄されるまで独自の**識別値**⁹⁹を持っており、オブジェクトの同一性は、そのオブジェクトの識別値によって判定される。オブジェクトの識別値は id 関数で調べることができる。

例. id 関数による識別値の調査（先の例の続き）

```
>>> id(a)  Enter  ←変数 a が保持するオブジェクトの識別値を調べる
2089578189128      ←変数 a の値の識別値
>>> id(b)  Enter  ←変数 b が保持するオブジェクトの識別値を調べる
2089578223496      ←変数 b の値の識別値
```

この例から、変数 a, b が保持するリストは同じ要素で構成されるが、実体としては別のものであることがわかる。

オブジェクトに対する識別値は、処理系がそのオブジェクトを生成する時点で決める。

先の例で変数 a にリストが割り当てられていたが、更に c = a として変数 c に変数 a の値を割り当てると、変数 c は変数 a と同一のリストを指し示す。

例. 「=」によって割り当てられたリストの識別値を調べる（先の例の続き）

```
>>> c = a  Enter  ←変数 a のリストを変数 c に割り当てる
>>> id(a)  Enter
2089578189128      ←変数 a の値の識別値
>>> id(c)  Enter  ←変数 c が保持するオブジェクトの識別値を調べる
2089578189128      ←変数 a の値の識別値と同じ
>>> a is c  Enter  ←変数 a, c の同一性を調べる
True              ←判定結果
```

2.6.4.6 値の型の判定

先の「2.4.12 型の検査」(p.52) で値の型を検査する方法について解説したが、実際の条件判定の処理において型の判定を行うための具体的な方法には次に示すような複数のものがある。

- 1) type(値) is 型名
- 2) type(値) == 型名
- 3) isinstance(値, 型名)

上記 3 つの判定方法における処理時間はどれも概ね同様であるが、2) の方法が若干遅く、3) の方法が若干早い傾向がある。

⁹⁹CPython 実装では、識別値はそのオブジェクトを格納しているメモリのアドレスである。

2.7 入出力

コンピュータのプログラムは各種の装置（デバイス）から入力を受け取って情報処理を行い、処理結果を各種の装置に出力する。入出力のための代表的な装置としては、

- ディスプレイ
- キーボード
- ファイル（ディスク）

が挙げられる¹⁰⁰。特にディスプレイとキーボードは常に使用可能なデバイスであることが前提とされている。このため、ディスプレイとキーボードはそれぞれ**標準出力**、**標準入力**と呼ばれている。

ここでは、これら 3 種類のデバイスに対する入出力の方法について説明する。

2.7.1 標準出力

通常の場合、標準出力はディスプレイを示している。これまでの解説にも頻繁に使用してきた `print` 関数は標準出力に対して出力するものであり、引数として与えた値を順番に標準出力に出力する。`print` 関数は任意の個数の引数を取る。

2.7.1.1 出力データの書式設定

表示桁数や表示する順番などの書式設定を施して出力をする際には、出力対象のデータ列（値の並び）を書式編集して一旦文字列にしてから出力することが一般的である。

■ 書式設定の方法 (1) `format` メソッド

文字列に対する `format` メソッドを用いて書式編集をすることができる。具体的には `{}` を含む文字列に対して `format` メソッドを実行すると、`format` の引数に与えた値が `{}` の部分に埋め込まれる。この `{}` を `format` メソッドの**プレースホルダ**（placeholder）と呼ぶ。

例. プレースホルダへの値の埋め込み

```
>>> s = '{}', {}, {}'.format('one', 'two', 'three')  Enter  ←書式編集
>>> print(s)  Enter  ←編集結果の確認
one, two, three  ←編集されて出力された結果
```

`{}` の中に埋め込みの順番を表すインデックス（0 で始まる整数）を与えると、`format` の引数を埋め込む順番の制御ができる。

例. 埋め込み位置の制御

```
>>> s = '{2}', {1}, {0}'.format('one', 'two', 'three')  Enter  ←書式編集
>>> print(s)  Enter  ←編集結果の確認
three, two, one  ←編集されて出力された結果
```

`{}` の中には更に、埋め込むデータの型（表 18）と桁数（長さ）を

:`[桁数]` 型

の形式で指定することができる。（桁数は省略可能）

表 18: `format` メソッドの書式設定に指定するデータの型（一部）

型	解 説	型	解 説	型	解 説	型	解 説
d	10 進整数	x	16 進整数	o	8 進整数	b	2 進整数
f	小数点数（暗黙で小数点以下 6 桁）	s	文字列				

例. 埋め込むデータの型と桁数の指定（文字列型）

```
>>> s = '|{2:7s}|{1:7s}|{0:7s}|'.format('one', 'two', 'three')  Enter  ←書式編集
>>> print(s)  Enter  ←編集結果の確認
|three |two  |one  |  ←編集されて出力された結果
```

¹⁰⁰この他にも重要なものとして、印刷装置（プリンタ）やネットワークインターフェース（NIC）、マウスなどがある。

この例では、'7s' の記述によってそれぞれ 7 桁の文字列になっている。

例. 埋め込むデータの型と桁数の指定（整数型）

```
>>> s = '|{2:7d}|{1:7d}|{0:7d}|'.format(1,2,3)  Enter  ←書式編集
>>> print(s)  Enter  ←編集結果の確認
|      3|      2|      1|  ←編集されて出力された結果
```

この例では、'7d' の記述によってそれぞれ 7 桁の整数になっている。

例. 埋め込むデータの型と桁数の指定（小数点数）

```
>>> s = '|{2:8.2f}|{1:8.2f}|{0:8.2f}|'.format(1.2,2.3,3.4)  Enter  ←書式編集
>>> print(s)  Enter  ←編集結果の確認
|   3.40|   2.30|   1.20|  ←編集されて出力された結果
```

この例では、'8.2f' の記述によってそれぞれ 2 桁の小数部を持つ合計 8 桁の小数点数になっている。

整数型と小数点数の桁数指定において、'7d' や '8.2f' の代わりに '07d' や '08.2f' のように左端にゼロを付けると、左にゼロを充填した表現が結果として得られる。

例. 左端のゼロの充填

```
>>> '{0:04d}'.format(2)  Enter  ←ゼロを充填した整数の処理
'0002'                  ←編集されて出力された結果
>>> '{0:08.2f}'.format(2)  Enter  ←ゼロを充填した小数点数の処理
'00002.00'              ←編集されて出力された結果
```

更に、桁数指定の部分に <, >, ^ を使用することで「左寄せ」、「右寄せ」、「中央揃え」といった **アラインメント** が可能となる。（次の例）

例. 「左寄せ」「右寄せ」「中央揃え」

```
>>> '|{0:<10d}|'.format(2)  Enter  ←左寄せ
'|2          |'           ←結果
>>> '|{0:>10d}|'.format(2)  Enter  ←右寄せ
'|          2|'           ←結果
>>> '|{0:^10d}|'.format(2)  Enter  ←中央揃え
'|      2      |'         ←結果
```

16 進、8 進、2 進表現の例を次に示す。

例. 16 進、8 進、2 進表現

```
>>> '16 進){:x}, 8 進){:o}, 2 進){:b}'.format(255,255,255)  Enter
'16 進)ff, 8 進)377, 2 進)11111111'  ←書式編集の結果
```

■ 書式設定の方法 (2) f-string を用いる方法

Python 3.6 から **フォーマット済み文字列リテラル** (f-string : formatted string literal) が導入された。これにより、書式の記述の中に変数名が記述できるなど、format メソッドと同じ処理がより簡単に実現できる。例えば次のような例について考える。

例. f-string 中に変数名を直接記述する

```
>>> v1='one'; v2='two'; v3='three'  Enter  ← 3 つの変数 v1, v2, v3 を用意
>>> s = f'{v1}, {v2}, {v3}'  Enter  ←プレースホルダに変数名を埋め込んで文字列を生成
>>> print(s)  Enter  ←内容確認
one, two, three  ←結果
```

この例にあるような、接頭辞 'f' を持つ文字列が f-string である。f-string は書式設定のための表現であり、format メソッドで行うような書式記述のプレースホルダ内に変数名や式を直接的に記述することができる。f-string 自体は書式設定の結果の文字列となる。

以下に、f-string による書式設定の例をいくつか示す。

例. f-string による文字列の書式設定 (先の例の続き)

```
>>> print( f' |{v1:7s}|{v2:7s}|{v3:7s}|' ) Enter ←文字列の桁数指定
|one      |two      |three  | ←結果
```

例. f-string による整数値の書式設定

```
>>> n1=1; n2=2; n3=3 Enter ← 3 つの変数 n1, n2, n3 に整数値を用意
>>> print( f' |{n1:7d}|{n2:7d}|{n3:7d}|' ) Enter ←整数値の桁数指定
|      1|      2|      3| ←結果
```

例. f-string による浮動小数点数の書式設定

```
>>> n1=1.2; n2=2.3; n3=3.4 Enter ← 3 つの変数 n1, n2, n3 に浮動小数点数を用意
>>> print( f' |{n1:8.2f}|{n2:8.2f}|{n3:8.2f}|' ) Enter ←浮動小数点数の桁数指定
|    1.20|    2.30|    3.40| ←結果
```

例. 数値の書式設定におけるゼロの充填

```
>>> n = 2 Enter ←変数 n に数値を用意
>>> f'{n:04d}' Enter ←ゼロを充填した整数表記
'0002' ←結果
>>> f'{n:08.2f}' Enter ←ゼロを充填した浮動小数点数の表記
'00002.00' ←結果
```

例. アラインメント (先の例の続き)

```
>>> f'|{n:<10d}|' Enter ←左寄せ
'|2          |' ←結果
>>> f'|{n:>10d}|' Enter ←右寄せ
'|          2|' ←結果
>>> f'|{n:^10d}|' Enter ←中央揃え
'|      2      |' ←結果
```

例. 16 進, 8 進, 2 進表現

```
>>> n = 255 Enter
>>> f'16 進){n:x}, 8 進){n:o}, 2 進){n:b}' Enter ←書式編集
'16 進)ff, 8 進)377, 2 進)1111111' ←結果
```

Python3.8 の版から f-string のプレースホルダに **{変数名=}** あるいは **{式=}** という記述が可能となった。
(次の例参照)

例. f-string の新しい書き方 (Python3.8 から)

```
>>> a=1; b=2; c=3 Enter ←変数に値を設定
>>> f'{a=}, {b=}, {c=}, {a+b+c=}' Enter ←変数名や式の内容を確認できる書き方
'a=1, b=2, c=3, a+b+c=6' ←結果
```

変数名や式の記述と、それに対応する値が表現された文字列が得られている。

■ 書式設定の方法 (3) '%' 演算子を用いる方法

これは C や Java における書式編集に似た方法である。先に説明した format メソッドによる方法と考え方が似ており、文字列中の '%' で始まる表記の場所 ('%' 演算子における **プレースホルダ**) に値を埋め込む方法である。

例. 文字列データの埋め込み (1)

```
>>> s = '|%7s|%7s|%7s|' % ('one','two','three') Enter ←書式編集
>>> print(s) Enter ←編集結果の確認
|    one|    two|   three| ←編集されて出力された結果
```

この例では、'%7s' の部分に後方のタプルの要素の値が 7 桁の文字列として埋め込まれている。基本的には右寄せの配置となるが、次の例のように '%' の後の数を負の値にすると左寄せの配置となる。

例. 文字列データの埋め込み (2)

```
>>> s = '|%-7s|%-7s|%-7s|' % ('one', 'two', 'three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|one      |two      |three |    ←編集されて出力された結果
```

このように '%' の後に表示桁数とデータタイプを指定する。データタイプとしては表 18 (p.103) のものが概ね使用できる。ただし、 '%' 演算子における書式編集の方法は、format メソッドや f-string の場合とは厳密には異なる (2 進編集の 'b' が使えないなど) ので注意すること。

■ 文字列のアラインメント

文字列に対する左寄せ、右寄せ、中央揃えのための ljust, rjust, center といったメソッドがある。

書き方: 文字列.ljust(長さ), 文字列.rjust(長さ), 文字列.center(長さ)

与えられた「文字列」を指定した「長さ」でアラインメントする。

例. 文字列のアラインメント

```
>>> s = 'abc'  Enter    ←この文字列 s を…
>>> s.ljust(7)  Enter    ←総長さ 7 で左寄せする
'abc      '        ←結果
>>> s.rjust(7)  Enter    ←総長さ 7 で右寄せする
'      abc'        ←結果
>>> s.center(7) Enter    ←総長さ 7 で中央揃えする
'   abc   '        ←結果
```

このように、アラインメントの結果として適切な個数の空白文字が充填される。

2.7.1.2 sys モジュールによる標準出力の扱い

print 関数による方法とは別に、sys モジュールを用いて標準出力に出力することもできる。

《 sys.stdout 》

オブジェクト sys.stdout は標準出力を示すオブジェクトである。このオブジェクトに対して write などのメソッドを使用して出力処理ができる。

標準出力への出力: sys.stdout.write(文字列オブジェクト)

標準出力に対して文字列を出力する。出力処理が正常に終了すると、出力した文字数^{*} が返される。

sys の使用に先立って、sys モジュールを読み込んでおく必要がある。

^{*} バイナリ形式で出力する場合は、出力バイト数が戻り値となる。標準出力にバイナリデータを出力する方法については後の「2.7.6 標準入出力でバイナリデータを扱う方法」(p.119) で解説する。

print 関数で出力すると行末で改行されるが、この方法による出力では自動的に改行処理はされない。(次の例を参照)

例. sys.stdout に対する出力

```
>>> import sys  Enter    ← sys モジュールの読み込み
>>> n = sys.stdout.write('abcd')  Enter    ←出力処理の実行
abcd>>>          ←出力結果
```

出力後は改行されずにプロンプト '>>>' が直後に表示されている。また write メソッドの実行後、出力したデータの長さ (文字数) が返される。

例. 戻り値の確認 (先の例の続き)

```
>>> n  Enter    ←出力バイト数の確認
4      ← 4 文字出力されたことが確認できる
```

表示の最後で改行するには、次のように文字列の最後にエスケープシーケンス '\n' を付ける。

例. write による出力後の改行方法（先の例の続き）

```
>>> n = sys.stdout.write('abcd¥n')  ←出力の最後で改行
abcd      ←表示の最後に改行されている
>>> n  ←戻り値の確認
5         ←5文字出力されたことが確認できる（改行コードも含む）
>>> n = sys.stdout.write('ab¥tcd¥n')  ←タブも出力可能
ab        cd      ←タブの表示と改行処理がされている
>>> n  ←戻り値の確認
6         ←6文字出力されたことが確認できる（タブ、改行コードも含む）
```

■ sys.stdout のエンコーディング設定

io モジュールを使用することで、sys.stdout に write メソッドで文字列を出力する際のエンコーディングを指定することができる。

例. sys.stdout のエンコーディングを utf-8 にする

```
import sys, io                # モジュールの読み込み
sys.stdout = io.TextIOWrapper( sys.stdout.buffer, encoding='utf-8' )
```

このように、io モジュールの TextIOWrapper オブジェクトを sys.stdout に設定する。この際、キーワード引数 'encoding=' にエンコーディングを指定する。

2.7.2 標準入力

通常の場合、標準入力はキーボードを示しており、ユーザからのキーボード入力を取得することができる。

2.7.2.1 input 関数による入力の取得

input 関数を呼び出すと、標準入力から 1 行分の入力を読み取って、それを文字列として返す。¹⁰¹

《 input 関数による入力の取得 》

書き方： input(プロンプト文字列)

input 関数を呼び出すと、「プロンプト文字列」を標準出力に表示して入力を待つ。1 行分の入力と改行入力 により、その 1 行の内容が文字列として返される。

input 関数を用いたサンプルプログラム test05-1.py を次に示す。

プログラム：test05-1.py

```
1 # coding: utf-8
2 x = input('入力x> ')
3 y = input('入力y> ')
4 print( x+y )
```

このプログラムを実行すると、

入力 x>

と表示され、プログラムは入力を待つ。続けて実行した様子を次に示す。

実行例.

```
入力 x> 2  ←「2」と入力
入力 y> 3  ←「3」と入力
23      ←出力
```

x+y の計算結果が '23' として表示されている。これは入力された値が文字列型であることによる。

次にサンプルプログラム test05-2.py について考える。

¹⁰¹ getpass モジュールを使用すると、パスワード用の秘匿入力ができる。その場合は、「from getpass import getpass」としてモジュールを読み込んで、getpass(プロンプト文字列) として入力取得する。

プログラム：test05-2.py

```
1 # coding: utf-8
2 x = int( input('入力x> ') )
3 y = int( input('入力y> ') )
4 print( x+y )
```

これは、input 関数の戻り値を int 関数によって整数型に変換している例である。このプログラムを実行した例を次に示す。

実行例.

```
入力 x> 2  [Enter]  ← 「2」と入力
入力 y> 3  [Enter]  ← 「3」と入力
5          ←出力
```

x+y の計算結果が 5 として表示されている。

2.7.2.2 sys モジュールによる標準入力の扱い

input 関数による方法とは別に、sys モジュールを用いて標準入力から入力することもできる。

《 sys.stdin 》

オブジェクト sys.stdin は標準入力を示すオブジェクトである。このオブジェクトに対して read や readline などのメソッドを使用して入力を取得することができる。

標準入力からの入力： sys.stdin.readline()

標準入力から 1 行分の入力を取得する。入力処理が正常に終了すると、取得したデータを文字列型のデータとして返す。

sys の使用に先立って、sys モジュールを読み込んでおく必要がある。

input 関数で入力する場合と異なり、この方法による入力では行末の改行コードも取得したデータに含まれる。(次の例を参照)

例. sys.stdin からの入力

```
>>> import sys  [Enter]  ← sys モジュールの読み込み
>>> s = sys.stdin.readline() [Enter]  ←入力を開始
abcde [Enter]  ← 1 行分のデータを入力
>>> s [Enter]  ←戻り値の確認
'abcde\n'  ←行末の改行コードも含まれている
```

■ sys.stdin のエンコーディング設定

io モジュールを使用することで、sys.stdin から文字列を入力する際のエンコーディングを指定することができる。

例. sys.stdin のエンコーディングを utf-8 にする

```
import sys, io  # モジュールの読み込み
sys.stdin = io.TextIOWrapper( sys.stdin.buffer, encoding='utf-8' )
```

このように、io モジュールの TextIOWrapper オブジェクトを sys.stdin に設定する。この際、キーワード引数 'encoding=' にエンコーディングを指定する。

注) 標準入力からバイナリデータを読み込む方法については後の「2.7.6 標準入出力でバイナリデータを扱う方法」(p.119) で解説する。

2.7.3 ファイルからの入力

ファイルからデータを読み込むにはファイルオブジェクトを使用する。ファイルオブジェクトはディスク上のファイルを示すものである。すなわち、ファイルからの入りに先立って open 関数を使用してファイルを開き、そのファイルに対応するファイルオブジェクトを生成しておく。以後はそのファイルオブジェクトからファイルのデータ(中身)

を取得することになる。

《 ファイルのオープン 》

open 関数を使用してファイルを開く

書き方： open(パス, モード)

「パス」は開く対象のファイルのパスを表す文字列型オブジェクトである。「モード」はファイルの開き方に関する設定であり、入力用か出力用か、あるいはテキスト形式かバイナリ形式かの指定をするための文字列型オブジェクトである。ファイルのオープンが成功すると、そのファイルのファイルオブジェクトを返す。

モード：

- r: 読取り用（入力用）にファイルを開く w: 書き込み用（出力）にファイルを開く
- a: 追記用（出力用）にファイルを開く x: 出力用にファイルを新規作成する、(既存の場合はエラー)
- r+: 入出力両用にファイルを開く、

ファイルは通常はテキスト形式として開かれるが、上記モードに **b** を書き加えるとバイナリ形式の扱いとなる。これに関しては後の「2.7.3.3 テキストファイルとバイナリファイルについて」(p.112) で更に詳しく解説する。

バイナリデータは Python 処理系では **バイト列** として扱われる。バイト列の扱いについては「2.7.3.4 バイト列の扱い」(p.112) で解説する。

open 関数が返すファイルオブジェクトの型（クラス）は入力用、出力用で異なる。更にテキストかバイナリかによっても異なる。

表 19: open 関数が返すファイルオブジェクトのクラス（一部）

	テキスト	バイナリ
入力用	TextIOWrapper	BufferedReader
出力用	(同上)	BufferedWriter

2.7.3.1 ファイル、ディレクトリのパスについて

パス (path) はファイルシステムにおけるファイルやディレクトリ（フォルダ）の場所を示すものであり、「/」や「¥」といった記号¹⁰² でディレクトリの階層関係を記したものである。

例. Windows でのパスの表記

C:¥Users¥katsu¥test1.txt
→ 「C ドライブのフォルダ Users の中のフォルダ katsu の中にあるファイル test1.txt」を意味する。

例. UNIX 系 OS (macOS, Linux など) でのパスの表記

/usr/home/katsu/test1.txt
→ 「ルートディレクトリの下にあるディレクトリ usr の下のディレクトリ home の下のディレクトリ katsu の下にあるファイル test1.txt」

■ ルートディレクトリ、カレントディレクトリ

UNIX 系 OS でのパスの表記において左端に「/」があれば、それはファイルシステムの**ルートディレクトリ**（最上位のディレクトリ）を意味する。また Windows でのパスの表記において左端に「c:¥」があれば、それは「ドライブ C」の**ルートフォルダ**（最上位のフォルダ）を意味する。左端にルートディレクトリ（ルートフォルダ）の記述があるパスを**絶対パス**と言う。

多くの場合、コンピュータのアプリケーションプログラムや言語処理システムには特定のディレクトリが**カレントディレクトリ**として設定されており、それが**暗黙のディレクトリ**となる。すなわち、ルートディレクトリの記述が無いパス（**相対パス**と言う）の表記では、カレントディレクトリがパスの起点となる。例えば、カレントディレクトリが '/usr/local' として設定されている場合、相対パス 'etc/dat1.txt' が示すものは

'/usr/local/etc/dat1.txt'

である。

¹⁰²使用するフォントによっては「¥」はバックスラッシュ「\」として表示される。

■ パスの表記における '.' と '..'

パスの表記における '.' と '..' はそれぞれ「現在のディレクトリ」「1つ上のディレクトリ」を意味する。例えば '..../file1.txt' というパスの記述は、「カレントディレクトリの1つ上のディレクトリにあるファイル 'file1.txt'」を意味する。

パスの記述は特殊な文字を含む場合があるので、パスを文字列として記述する際、**raw 文字列**とするのが安全である。例えば「C:¥Users¥katsu¥test1.txt」というパスは「r'C:¥Users¥katsu¥test1.txt」と記述すると良い。

2.7.3.2 扱うファイルのエンコーディング、改行コードの指定

テキスト形式でファイルを開く場合は、対象のファイルのエンコーディングに注意する必要がある。Python の処理系がエンコーディングとして `shift_jis` の扱いを前提としている場合、`utf-8` など他のエンコーディングのファイルを読み込むとエラーが発生することがある。従って、ファイルをテキスト形式で開く場合は、次の例のように `encoding` を指定して、読み込むファイルのエンコーディングを指定しておく安全である。

例. `utf-8` のテキストファイルを開く場合

```
open(ファイルのパス,'r',encoding='utf-8')
```

Python で扱えるエンコーディングは「2.1.1 プログラム中に記述するコメント」(p.6) で説明した表 1 のものを指定する。

テキストファイルの改行コードは OS 毎に異なることがあり、Windows 環境では `CR`+`LF` (`0x0D`, `0x0A`) が、Unix 系 (Linux, macOS など) では `LF` が標準的である。この違いはテキストファイルの出力時に問題を起こすことがあるので注意が必要である。後に説明するファイル出力においては、`open` 関数にキーワード引数 '`newline=改行コード`' を与えると安全である。この場合の「改行コード」には

```
None, '', '¥n', '¥r', '¥r¥n'
```

といったものが指定できる。この際、`None` (デフォルト値) を指定すると、当該 OS の標準的な改行コード¹⁰³ となり、空文字列 '' を指定すると `¥n` となる。

ファイルからのデータ入力の例：

テキストファイルから 1 行ずつデータを取り出すプログラム `test06-1.py` を例示する。

プログラム：`test06-1.py`

```
1 # coding: utf-8
2 f = open('test06-1.txt','r')
3
4 while True:
5     s = f.readline()
6     if s:
7         print(s)
8     else:
9         break
10
11 f.close()
```

基本的な考え方：

このプログラムはテキストファイル `test06-1.txt` を開き、それをファイルオブジェクト `f` としている。このファイルオブジェクトに対して `readline`¹⁰⁴ メソッドを使用してデータを 1 行ずつ取り出して変数 `s` に与え、それを `print` 関数で表示している。ファイルの内容を全て読み終わると、次回 `readline` 実行時にデータが得られない¹⁰⁵ ので、`break` により `while` を終了する。

¹⁰³os モジュールの `os.linesep` に設定されている。

¹⁰⁴`readline` メソッドは、開かれているファイルのモードによって返す値の型が異なり、テキストモードで開いているときは文字列型で、バイナリモードで開いているときはバイト列（「2.7.3.4 バイト列の扱い」(p.112) 参照）で返す。

¹⁰⁵変数 `s` に空文字列 '' がセットされる。

テキストファイル：test06-1.txt

```
1 1行目
2 2行目
3 3行目
```

プログラム test06-1.py を実行すると次のように表示¹⁰⁶ される。

```
1 行目
2 行目
3 行目
```

この実行例では余分に改行された形で表示されている。これは、`readline` が改行コード¹⁰⁷ も含めて取得するため、変数 `s` にセットされる文字列オブジェクトの末尾にも改行コードが含まれるからである。

文字列オブジェクトの行末の改行コードを削除するには `rstrip` メソッド¹⁰⁸ を使用する。

《 改行コードの削除 》

書き方： 対象文字列.`rstrip()`

「対象文字列」の末尾にある改行コードや余分な空白文字を削除する。

`rstrip` メソッドを用いた形に修正したプログラム test06-2.py を示す。

プログラム：test06-2.py

```
1 # coding: utf-8
2 f = open('test06-1.txt', 'r')
3
4 while True:
5     s = f.readline().rstrip()
6     if s:
7         print(s)
8     else:
9         break
10
11 f.close()
```

このプログラムを実行すると次のように表示される。

```
1 行目
2 行目
3 行目
```

ファイルからの読み込みが終われば、そのファイルを閉じる。

《 ファイルのクローズ 》

書き方： ファイルオブジェクト.`close()`

開かれている「ファイルオブジェクト」を閉じる。

テキスト読み込みに伴うエンコーディングの不具合について：

テキスト形式の入力データのエンコーディングによっては、入力処理において問題を起こす場合がある。

(次の実行例)

```
1 隋橋岬
2 隋橋岬
3 隋橋岬
```

¹⁰⁶この時に不可解な文字列が表示されることがあるが、対処法については後で説明する。

¹⁰⁷バイトデータとしての実際のコードは OS 毎に異なる。例。Windows: `CR LF`，macOS, Linux: `LF`
使用している処理系の改行コードは `os` モジュールの `os.linesep` に設定されている。

¹⁰⁸`rstrip` メソッドは、対象文字列の改行文字だけでなく、右端の余分な空白文字も削除する。これと類似のものに `lstrip` メソッドも存在し、これは文字列の左端の余分な空白文字を削除する。

ファイルのデータをテキスト形式として扱う際、エンコーディングが正しく識別されない場合にはこのような現象¹⁰⁹が起こることがある。解決策としては、先に述べたように open 関数にキーワード引数 'encoding=' を与えるのが基本的であるが、別の方法として、テキストファイルの読み込みにおいてもバイナリ形式でファイルを扱うということが挙げられる。バイナリ形式としてファイルからデータを入力すると、それらはデータ型とは無関係な**バイト列**とみなされる。(p.112 「2.7.3.4 バイト列の扱い」参照)

バイト列として得られたデータを、プログラム側で明に各種のデータ型の値に変換することでより安全な処理が実現できる。すなわち、バイト列として得られたデータを、正しいエンコーディングの扱いを指定した上で文字列型のオブジェクトに変換し、更にそれらを目的のデータ型のオブジェクトに変換するという方法である。

次のプログラム test06-3.py について考える。

プログラム：test06-3.py

```
1 # coding: utf-8
2 f = open('test06-1.txt', 'rb')
3
4 while True:
5     s = f.readline().decode('utf-8').rstrip()
6     if s:
7         print(s)
8     else:
9         break
10
11 f.close()
```

このプログラムの5行目の部分に見られる decode メソッドは、バイト列のデータに対するメソッドであり、引数に指定したエンコーディングで文字列型に変換する。

このような形でファイルからデータを入力すると、多バイト系の文字も正しく処理される。

2.7.3.3 テキストファイルとバイナリファイルについて

ファイルシステム上のファイルは8ビット表現の2進数を1つの単位(1バイト)とする**バイト列**として構成されたものであるが、コンピュータのデータファイルは**テキストファイル**と**バイナリファイル**の2種類に区別して扱うことが多い。テキストファイルは**アスキー文字**を始めとする可読な文字¹¹⁰と、表示の制御のための**制御文字**から成るもの¹¹¹であり、それらに該当しないバイト値を含むファイルはバイナリファイルとして扱われる。

Python 言語処理系では、可読な文字と表示のための制御文字から成るデータを**文字列型**(str 型)の値として扱う。またそのようなデータを**テキストデータ**(あるいは「テキスト形式データ」と呼ぶことがある。

文字列型のデータとしてファイルの入出力を行う場合は open 関数の第2引数に与える**モード**として 'r' (入力時)や 'w' (出力時)を指定する。また、モードに 'rb' (入力時)や 'wb' (出力時)を指定すると、対象のファイルをバイナリファイルとして扱う。バイナリファイルに対する入出力は、次に説明する**バイト列**(bytes 型)の値として行う。

テキストファイルを敢えてバイナリファイルとして扱う(open 関数の第2引数に 'rb' や 'wb' を与える)ことも可能である¹¹²が、その場合は入出力の値を文字列(str 型)としてではなく、バイト列(bytes 型)の値として扱う。

2.7.3.4 バイト列の扱い

バイト列は文字列型や数値型とは異なるオブジェクトであり、ファイル入出力や通信に用いる際の基本的なオブジェクトである。これは、テキスト形式ではないいわゆる**バイナリデータ**を扱う場合に用いられるデータ型である。バイト列のデータ型は bytes である。

コンピュータで扱うデータは、記憶資源上の実体としては(メモリに格納される実体としては)数値、文字列に限らず全てバイト列であり、処理系が扱う際には、それらバイト列を数値や文字列といった目的の型に解釈して処理を行う。例えば文字列型オブジェクトを用いると、多バイト系文字列も次の例のように正しく表示することができる。

¹⁰⁹いわゆる「文字化け」

¹¹⁰日本語などの多バイト文字も含む。

¹¹¹1 バイトで表現できる可読文字と制御文字に関しては巻末付録「J.2 アスキーコード表」(p.466)を参照のこと。

¹¹²コンピュータで扱うあらゆるデータは元来**バイナリデータ**である。

例. 多バイト文字列（日本語）の扱い

```
>>> a = '日本語'  Enter  ←日本語文字列の作成
>>> a  Enter  ←内容の確認
'日本語'  ←正しく表示されている
```

これは Python 処理系が多バイト系文字列データの文字コード体系（エンコーディング）を正しく解釈して処理しているからであるが、この文字列はバイト列としては（記憶資源上の内部表現としては）、

```
e6 97 a5 e6 9c ac e8 aa 9e      (16 進数表現)
```

というバイト値¹¹³の列であり、ファイルとして保存する場合もディスクにはこのようなバイト値の並びとして記録されている。すなわち、Python を始めとする処理系は日本語など多バイト文字をディスプレイに表示する際、正しい記号として表示するように処理系内部で処理している。

【バイト列に関すること】

Python では、通信やファイル入出力において各種のオブジェクトの内容を実際のデバイスとやりとりする際に、それらをバイト列として扱っている。デバイスに出力する際にはデータをバイト列に変換して出力し、デバイスから入力する際にはバイト列として入力した後に適切なデータ型に変換する。先の例で扱った日本語の文字列型オブジェクトも明にバイト列に変換することができる。（次の例参照）

例. 文字列をバイト列に変換する（先の例の続き）

```
>>> b = a.encode('utf-8')  Enter  ←バイト列に変換
>>> b  Enter  ←内容の確認
b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e'  ←バイト列データ
```

このように encode メソッドを使用することで、文字列型オブジェクトをバイト列に変換することができる。encode メソッドの引数には、元の文字列のエンコーディングを与える。

バイト列として表現されている多バイト文字列のデータを文字列型に変換するには、そのバイト列に対して decode メソッドを用いる。

例. バイト列を文字列に変換する

```
>>> b = b'\xe6\x96\x87\xe5\xad\x97'  Enter  ←バイト列
>>> b.decode('utf-8')  Enter  ←それを utf-8 の文字列に変換
'文字'  ←変換結果
```

《 バイト列⇄多バイト系文字列の変換 》

- バイト列⇒文字列の変換： 対象バイト列.decode(エンコーディング) 戻り値は文字列型オブジェクト
- 文字列⇒バイト列の変換： 対象文字列.encode(エンコーディング) 戻り値はバイト列

応用：

encode, decode メソッドを組み合わせることで「文字列⇒バイト列⇒文字列」と変換することで、エンコーディングの変換処理が実現できる。

参考：エンコーディングを指定してバイト列を文字列に変換するには str を使う方法もある。

書き方： str(バイト列, エンコーディング)

例. バイト列を文字列に変換する：その2（先の例の続き）

```
>>> str( b, 'utf-8' )  Enter  ← utf-8 の文字列に変換
'文字'  ←変換結果（decode メソッドと同じ結果）
```

■ バイト列の作成方法

bytes コンストラクタでバイト列を作成する方法について説明する。

書き方： bytes(整数値のリスト)

¹¹³0~255 の整数値（16 進数では 00~ff）

「整数値のリスト」¹¹⁴ の要素は 0～255 の整数である。

例. 整数値のリストをバイト列に変換する

```
>>> b = bytes([230,150,135,229,173,151])  Enter  ←整数のリストをバイト列に変換
>>> b  Enter  ←内容確認
b'\xe6\x96\x87\xe5\xad\x97'  ←バイト列
>>> b.decode('utf-8')  Enter  ←文字列に変換
'文字'  ←変換結果
```

次のような書き方で文字列をバイト列に変換することもできる。

書き方: bytes(文字列, エンコーディング)

「文字列」を「エンコーディング」に従ってバイト列に変換する。これは encode メソッドと同じ結果となる。

例. 文字列をバイト列に変換する

```
>>> bytes('日本語', 'utf-8')  Enter  ←文字列をバイト列に変換
b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e'  ←変換結果
```

注意) bytes 型のデータは作成後に変更することができない。(イミュータブルである) ミュータブル(変更可能)なバイト列に関しては「4.13 編集可能なバイト列: bytearray」(p.317)で解説する。

2.7.3.5 バイト列のコード体系を調べる方法

ファイルや通信デバイスから入力されたバイトデータを多バイト系文字列として解釈する場合に、それを表現するためのエンコーディングが分からないことがある。そのような場合は、chardet¹¹⁵ ライブラリを用いてかなり正確にエンコーディングを識別することができる。

例. chardet ライブラリによるエンコーディングの識別

```
>>> import chardet  Enter  ←ライブラリの読み込み
>>> s = '私はPython'  Enter  ←文字列の設定
>>> b = s.encode('utf-8')  Enter  ←バイト列への変換
>>> d = chardet.detect(b)  Enter  ←エンコーディングの識別処理
>>> print(d['encoding'])  Enter  ←結果を調べる
utf-8  ←utf-8 であることがわかる。
```

このように detect 関数の引数に調べたいバイト列を与えると、識別結果が辞書オブジェクトとして返される。結果の辞書オブジェクトのキー 'encoding' に対する値として、エンコーディングの名称が得られる。

2.7.3.6 指定したバイト数だけ読み込む方法

バイナリファイルとして開いているファイル f から指定したバイト数だけ読み込むには read メソッドの引数にバイト数を整数で与える。

例. バイナリファイル f から 256 バイト読み込む

```
b = f.read(256)
```

この例では、f から読み取った 256 バイト分のデータが b に格納される。

2.7.3.7 ファイルの内容を一度で読み込む方法

入力用のファイルオブジェクトに対して read メソッドを引数なしで実行すると、ファイルの内容を全て読み込むことができる。

例. ファイルの内容を全て読み込む

```
f = open(ファイル名, 'r', encoding='utf-8')
text = f.read()
f.close()
```

¹¹⁴ リストに限らずイテラブルであれば良い。

¹¹⁵ 詳しくはインターネットサイト <https://chardet.readthedocs.io/> を参照のこと。PSF 版 Python におけるインストール方法に関しては「A.4 PIP によるライブラリ管理」を参照のこと。Anaconda の場合は、Anaconda Navigator でパッケージ管理を行う。

ただし、ファイルのデータサイズが大きい場合は注意が必要であり、行単位の読み込みをする等の工夫が必要になることがある。既存のファイルのサイズを予め調べる方法については「2.7.7.5 ファイルのサイズの取得」(p.121)を参照のこと。

応用：

テキストファイルの内容を read メソッドで一度に読み込んだ後、先に解説した `splitlines`¹¹⁶ を応用すると、テキストの各行を要素とするリストが得られる。

ファイルを開くための別の方法を「2.7.8 パス（ファイル、ディレクトリ）の扱い：その2 - `pathlib` モジュール」(p.123)で説明する。

2.7.3.8 ファイルをイテラブルとして読み込む方法

`open` 関数で開いたファイルはイテラブルなオブジェクト¹¹⁷ として扱うことができる。この場合にテキストファイルから取り出す1つの要素は、ファイルの行である。例えば、次に示すテキストファイル `dat1.txt` をイテラブルとして読み込むことを考える。

テキストファイル：dat1.txt

```
1 1行目
2 2行目
3 3行目
```

これを読み込む例を次に示す。

例. ファイルをイテラブルと見て読み込む

```
>>> f = open('dat1.txt', 'r', encoding='utf-8')  Enter    ←ファイルのオープン
>>> for s in f:  Enter    ←ファイル f をイテラブルとして繰り返しを始める
...     print( s.rstrip() )  Enter    ←取り出した要素を表示
...  Enter    ←繰り返しの記述の終了
1行目    ←取り出した要素を表示
2行目    ←取り出した要素を表示
3行目    ←取り出した要素を表示
>>> f.close()  Enter    ←ファイルのクローズ
```

ファイル `f` に対して `readline` メソッドで行を取り出す処理に似ているが、これはあくまで「イテラブル」としての扱いである。

課題. ファイルオブジェクトをイテラブルとして扱うことができることから、ファイルオブジェクト `f` に対して `list(f)` とすると何が得られるかについて考察し、その応用可能性について考えよ。

2.7.3.9 `readlines` メソッドによるテキストファイルの読み込み

先に説明した `readline` メソッドは、テキスト形式のファイルオブジェクトから内容を1行ずつ読み込む。これに対して `readlines` メソッドはテキストファイルの内容を全て読み込み、その各行を要素とするリストを返す。

例. `readlines` によるテキストファイルの読み込み（先の例の続き）

```
>>> f = open('dat1.txt', 'r', encoding='utf-8')  Enter    ←ファイルのオープン
>>> txtL = f.readlines()  Enter    ←ファイルの内容をすべて読み込んでリストにする
>>> print(txtL)  Enter    ←得られたリストを表示
['1行目\n', '2行目\n', '3行目\n']    ←リストの内容
>>> f.close()  Enter    ←ファイルのクローズ
```

得られたリストの各要素には改行コード「`\n`」も含まれる。

¹¹⁶p.42「`splitlines` メソッドによる行の分離」を参照のこと。

¹¹⁷「2.6.1.1 イテラブルなオブジェクト」(p.89)、「2.6.1.8 イテレータ」(p.92)を参照のこと。

2.7.3.10 データを読み込む際のファイル中の位置について

入力用のファイルオブジェクトに対して `tell` メソッドを実行すると、次にデータを読み込む位置（ファイル中でのバイト位置）が得られる。

例. ファイル中の読み取り位置を調べる（先の例の続き）

```
>>> f = open('dat1.txt','r',encoding='utf-8')  Enter    ←ファイルのオープン
>>> f.tell()  Enter    ←ファイルをオープンした直後の読み取り位置を調べる
0                                                    ←0 バイト目から読み取る状態
>>> f.readline().rstrip()  Enter    ←1 行目の読み込み
'1 行目'                                              ←読み込んだ内容
>>> f.tell()  Enter    ←次の読み取り位置を調べる
9                                                    ←9 バイト目から読み取る状態
>>> f.readline().rstrip()  Enter    ←2 行目の読み込み
'2 行目'                                              ←読み込んだ内容
>>> f.tell()  Enter    ←次の読み取り位置を調べる
18                                                   ←18 バイト目から読み取る状態
>>> f.readline().rstrip()  Enter    ←3 行目の読み込み
'3 行目'                                              ←読み込んだ内容
>>> f.tell()  Enter    ←最終的なファイルの位置を調べる
27                                                   ←27 バイト目
>>> f.close()  Enter    ←ファイルのクローズ
```

`tell` メソッドは出力用のファイルオブジェクトに対して使用することもできる。その場合は「ファイル中での出力位置」が得られる。

2.7.4 ファイルへの出力

開かれたファイルに対してデータを出力することができる。ファイルのオープンとクローズについては先の「2.7.3 ファイルからの入力」のところで説明したとおりであり、ここではファイルオブジェクトに対する出力について解説する。

ファイルオブジェクトに対する出力には `write` メソッドを使用する。

《 write メソッド 》

書き方： ファイルオブジェクト.write(データ)

「ファイルオブジェクト」に対して「データ」を出力する。「データ」に与えるオブジェクトの型は、ファイルオブジェクトのモードによる。すなわち、ファイルオブジェクトがテキストモードのときは文字列型で、バイナリモードの場合はバイト列で与える。

このメソッドが正常に実行されると、出力したデータの長さ（テキストモードの場合は文字数、バイナリモードの場合はバイト数）が返される。

テキストデータ出力時のエンコーディング、改行コードの指定に関しては、先の「2.7.3.2 扱うファイルのエンコーディング、改行コードの指定」(p.110) を参照のこと。

2.7.4.1 print 関数によるファイルへの出力

`print` 関数もファイルへの出力の機能を持つ。

《 print 関数によるファイルへの出力 》

書き方： `print(..., file=ファイルオブジェクト)`

出力内容に続いてキーワード引数 `'file='` を与え、これに出力先のファイルオブジェクトを指定する。

2.7.4.2 writelines メソッドによる出力

ファイルオブジェクトに対して writelines メソッドを用いてデータを出力することができる。

《 writelines メソッドによるファイルへの出力 》

書き方： ファイルオブジェクト.writelines(リスト)

「リスト」の要素を順番に「ファイルオブジェクト」が示すファイルに出力する。

注意) 「リスト」の要素を1つの行としてテキストデータを出力する場合は、各要素の末尾に改行（`\n`）を付けること。

`write`, `print`, `writelines` をそれぞれ用いてファイルに出力するサンプルプログラム `writeTest03.py` を次に示す。

プログラム： `writeTest03.py`

```
1 # coding: utf-8
2 f = open('writeTest03.txt', 'w', encoding='utf-8')
3
4 print('writeメソッドを実行する直前のファイル中でのバイト位置:', f.tell() )
5 f.write('writeメソッドによる出力\n')
6
7 print('print関数を実行する直前のファイル中でのバイト位置:', f.tell() )
8 print('print関数による出力 ', file=f)
9
10 print('writelinesメソッドを実行する直前のファイル中でのバイト位置:', f.tell() )
11 L = ['writelinesによる連続出力：1行目\n', '2行目\n', '3行目\n']
12 f.writelines(L)
13
14 print('ファイルを閉じる直前のファイル中でのバイト位置:', f.tell() )
15 f.close()
```

これを実行すると次のようなファイル `writeTest03.txt` ができる。

出力されたファイル： `writeTest03.txt`

```
1 writeメソッドによる出力
2 print関数による出力
3 writelinesによる連続出力：1行目
4 2行目
5 3行目
```

また、各出力処理の直前での出力位置（ファイル中でのバイト位置）が標準出力に出力される。（次の例）

出力例

```
writeメソッドを実行する直前のファイル中でのバイト位置: 0
print関数を実行する直前のファイル中でのバイト位置: 34
writelinesメソッドを実行する直前のファイル中でのバイト位置: 62
ファイルを閉じる直前のファイル中でのバイト位置: 123
```

2.7.4.3 出力バッファの書き出し

`write` メソッドなどによるファイルへのデータの出力においては、出力内容は一旦**出力バッファ**に蓄えられ、それが一杯になった時点で実際にストレージに書き込まれる。そして出力バッファは空になり、次の出力内容を受け付けて蓄える。この他にも、ファイルが閉じられる際や、プログラム自体が終了（Python 処理系が終了）する時点でも出力バッファの内容がストレージに書き込まれる。出力バッファの内容を意図して（強制的に）ストレージに書き込むには、出力ファイルのファイルオブジェクトに対して `flush` メソッドを実行する。

書き方： ファイルオブジェクト.flush()

2.7.5 標準エラー出力

標準出力によく似た働きを持つ**標準エラー出力**というものがある。sys モジュールの `stderr` に対して `write` メソッドを実行すると、標準出力の場合と同様にディスプレイに出力結果が表示される。次のプログラム `test06-4.py` の動作について考える。

プログラム：test06-4.py

```
1 # coding: utf-8
2 # モジュールの読み込み
3 import sys
4
5 for i in range(50):
6     sys.stdout.write(str(i)+' ',')
7     if i > 0 and i % 10 == 0:
8         sys.stdout.write('\n')
9         sys.stderr.write(str(i)+' まで出力しました. \n')
```

これは 0 から 49 までの整数を標準出力に出力するプログラムであるが、値が 10,20,30,40 のときに標準エラー出力に対してメッセージ「～まで出力しました」を表示するものである。(次の実行例参照)

実行例 1. Windows 環境のコマンドシェル (cmd.exe) での実行

```
C:\Users\student>py test06-4.py
0,1,2,3,4,5,6,7,8,9,10,
10 まで出力しました.
11,12,13,14,15,16,17,18,19,20,
20 まで出力しました.
21,22,23,24,25,26,27,28,29,30,
30 まで出力しました.
31,32,33,34,35,36,37,38,39,40,
40 まで出力しました.
41,42,43,44,45,46,47,48,49,
```

この例では、標準出力と標準エラー出力が同じ働きをしているように見える。ただし、これら 2 種類の出力はそれぞれ別のものであり、**出力先のリダイレクト**¹¹⁸により、別の出力先に送り出すことができる。(次の実行例参照)

実行例 2. Windows 環境のコマンドシェル (cmd.exe) での実行

```
C:\Users\student>py test06-4.py 1> test06std.txt 2> test06err.txt
```

この処理の結果、標準出力への出力がファイル test06std.txt に、標準エラー出力への出力がファイル test06err.txt に書き込まれる。(次の例参照)

ファイル test06std.txt

```
1 0,1,2,3,4,5,6,7,8,9,10,
2 11,12,13,14,15,16,17,18,19,20,
3 21,22,23,24,25,26,27,28,29,30,
4 31,32,33,34,35,36,37,38,39,40,
5 41,42,43,44,45,46,47,48,49,
```

ファイル test06err.txt

```
1 10 まで出力しました.
2 20 まで出力しました.
3 30 まで出力しました.
4 40 まで出力しました.
```

■ 標準エラー出力の主な用途

ターミナル系のプログラム（標準入出力を基本的な UI とするプログラム）では、エラーメッセージや各種ログの出力（プログラム実行中の報告出力）などを、主たる出力とは区別する習慣がある。そのような場合に標準エラー出力が出力先として用いられる。

ターミナル系のプログラムでは図 6 に示すように、**標準的に**出力が 2 系統、入力が 1 系統存在している。

■ sys.stderr のエンコーディング設定

io モジュールを使用することで、sys.stderr に write メソッドで文字列を出力する際のエンコーディングを指定することができる。

¹¹⁸標準出力や標準入力とは通常ではディスプレイやキーボードに接続されているが、これらをファイルに接続することができる。

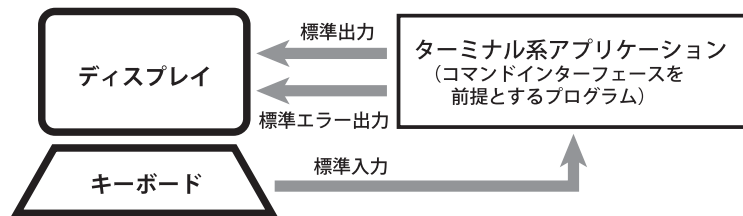


図 6: ターミナル系プログラムの入出力の概観

例. `sys.stderr` のエンコーディングを `utf-8` にする

```
import sys, io
sys.stderr = io.TextIOWrapper( sys.stderr.buffer, encoding='utf-8' )
```

このように、`io` モジュールの `TextIOWrapper` オブジェクトを `sys.stderr` に設定する。この際、キーワード引数 `'encoding='` にエンコーディングを指定する。

注) 標準エラー出力にバイナリデータを出力する方法については次の「2.7.6 標準入出力でバイナリデータを扱う方法」(p.119) で解説する。

■ `print` 関数による標準エラー出力への出力

`print` 関数にキーワード引数 `file=sys.stderr` を与えることで、標準エラー出力に出力することができる。例えば次のようなプログラム `test06-7.py` を考える。

プログラム: `test06-7.py`

```
1 import sys
2
3 print('標準出力')
4 print('標準エラー出力', file=sys.stderr)
```

これを次のようにして実行する。

実行例 3. Windows 環境のコマンドシェル (`cmd.exe`) での実行

```
C:\Users\student>py test06-7.py 1> test06std2.txt 2> test06err2.txt
```

この結果、標準出力と標準エラー出力への出力が次のような 2 つのファイルとして作成される。

ファイル `test06std2.txt`

```
1 標準出力
```

ファイル `test06err2.txt`

```
1 標準エラー出力
```

2.7.6 標準入出力でバイナリデータを扱う方法

`sys` モジュールを用いて標準入出力、標準エラー出力でバイナリデータを扱う方法を表 20 に示す。

表 20: 標準入出力、標準エラー出力でのバイナリデータの扱い

記述	解説
<code>sys.stdin.buffer.read(n)</code> <code>sys.stdin.buffer.read()</code>	標準入力からバイナリデータを <code>n</code> バイト読み込んで返す。 標準入力からバイナリデータをファイル終端 (EOF) まで読み込んで返す。
<code>sys.stdout.buffer.write(b)</code>	bytes オブジェクト <code>b</code> を標準出力へ出力し、 出力したバイト数を返す。
<code>sys.stderr.buffer.write(b)</code>	bytes オブジェクト <code>b</code> を標準エラー出力へ出力し、 出力したバイト数を返す。

2.7.7 パス（ファイル、ディレクトリ）の扱い：その1 - os モジュール

ここではパス（ファイル、ディレクトリ）に対する各種の操作について説明する。パスに対する操作をするには os モジュールを使用するので、次のようにして読み込んでおく。

```
import os
```

2.7.7.1 カレントディレクトリに関する操作

相対パスを指定してファイルの入出力を行う場合は**カレントディレクトリ**を基準とする。Python のプログラム実行時には、処理系を起動した際のディレクトリがカレントディレクトリとなるが、プログラムの実行時にこれを変更することができる。

■ カレントディレクトリの取得

os モジュールの `getcwd` メソッドを使用する。

例. カレントディレクトリを調べる

```
>>> import os      [Enter]    ←モジュールの読み込み
>>> os.getcwd()    [Enter]    ←カレントディレクトリを調べる
'C:\¥¥Users\¥¥katsu'      ←カレントディレクトリ
```

結果は文字列の形式で得られる。これは Windows における実行例であり、「¥」はエスケープされて「¥¥」となる。

■ カレントディレクトリの変更

os モジュールの `chdir` メソッドを使用する。

例. カレントディレクトリの変更（先の続き）

```
>>> os.chdir('.')  [Enter]    ←カレントディレクトリを変更（1つ上へ）
>>> os.getcwd()    [Enter]    ←カレントディレクトリを調べる
'C:\¥¥Users'       ←カレントディレクトリが変更されている
```

2.7.7.2 ホームディレクトリの取得

カレントユーザ¹¹⁹ のホームディレクトリは OS の**環境変数**が保持している。環境変数は os モジュールの辞書 `environ` から参照できる。これに関しては後の「4.25.4 環境変数の参照」（p.350）で解説するが、ここではホームディレクトリを取得する方法について例示する。

例. ホームディレクトリの取得（macOS の場合）¹²⁰

```
>>> import os      [Enter]    ←モジュールの読み込み
>>> print( os.environ['HOME'] )  ← OS の環境変数 HOME を出力
/Users/katsu       ←ホームディレクトリ
```

例. ホームディレクトリの取得（Windows の場合）

```
>>> import os      [Enter]    ←モジュールの読み込み
>>> home = os.environ['HOMEDRIVE'] + os.environ['HOMEPATH'] [Enter] ←ホームディレクトリの取得
>>> print( home )   [Enter]    ←ホームディレクトリの出力
C:\¥¥Users\¥¥katsu ←ホームディレクトリ
```

2.7.7.3 ディレクトリ内容の一覧 (1)

os モジュールの `listdir` 関数を使用する。

例. ディレクトリ内容の一覧（先の続き）

```
>>> os.listdir()   [Enter]    ←内容リストの取得
['file1.txt', 'a.exe', ... ]  ←実行結果
```

結果はリストの形式で得られる。この例の様に、`listdir` 関数の引数を省略するとカレントディレクトリの内容の一覧が得られるが、キーワード引数 `path='パス'` に対象のパス（ディレクトリ）を指定することもできる¹²¹。

¹¹⁹ログインしているユーザのこと。

¹²⁰Linux の場合も同様の方法を取ることができる。

¹²¹第一引数に文字列として対象のディレクトリを与えてもよい。

listdir 関数は、サブディレクトリを再帰的に検索しない。

2.7.7.4 ディレクトリ内容の一覧 (2)

os モジュールの walk 関数を使用すると、サブディレクトリを再帰的に検索する形で内容の一覧を取得する。

書き方: walk(対象ディレクトリ)

「対象ディレクトリ」の内容をもたらすジェネレータを返す。得られるジェネレータは次のような形式のタプルをもたらす。

(格納ディレクトリ, サブディレクトリのリスト, ファイルのリスト)

※ ジェネレータに関しては、後の「4.7 ジェネレータ」(p.297) で解説する。

例えば、カレントディレクトリに右図のような階層構造（`<...>` はディレクトリ）を持つディレクトリ exdir がある場合の walk 関数の実行例を次に示す。

```
<exdir>
+- <emp>
|
+- file1.txt
|
+- <subdir>
|   +- file2.txt
|   +- file3.txt
|   +- <subsubdir>
|       +- file4.txt
```

例. walk 関数の使用例 (先の例の続き)

```
>>> for x in os.walk('./exdir'): Enter ←内容取得の反復
...     print(x) Enter ←内容項目の出力
... Enter ← for 文の記述の終了
('./exdir', ['emp', 'subdir'], ['file1.txt']) ←↓内容項目の一覧
('./exdir/emp', [], [])
('./exdir/subdir', ['subsubdir'], ['file2.txt', 'file3.txt'])
('./exdir/subdir/subsubdir', [], ['file4.txt'])
```

この処理を応用して、ディレクトリ項目、ファイル項目を1つずつ取得するジェネレータを実装する例を巻末付録「I.3 os.walk 関数の応用例」(p.460) に示す。

2.7.7.5 ファイルのサイズの取得

os.path.getsize 関数で、既存のファイルのサイズを取得することができる。この関数の引数には、調査対象のファイルのパスを与える。

例. ファイルサイズの取得

```
>>> import os Enter ←モジュールの読み込み
>>> os.path.getsize('file1.txt') Enter ←ファイル'file1.txt'のサイズの取得
31 ← 31 バイト
```

2.7.7.6 ファイル、ディレクトリの検査

指定したパス p が存在するかどうかを検査するには os.path.exists(p) を実行する。

例. パスの存在検査 (先の例の続き)

```
>>> os.path.exists('file.txt') Enter ←存在するファイル file.txt の検査
True ←真 (存在する)
>>> os.path.exists('xxx') Enter ←存在しないファイル xxx の検査
False ←偽 (存在しない)
```

戻り値は真理値 (True / False) である。ディレクトリの存在を調べる場合も同様の方法で検査できる。

指定したパス p がファイルかどうかを検査するには os.path.isfile(p) を、ディレクトリかどうかを検査するには os.path.isdir(p) を実行する。どちらも戻り値は真理値である。

例. ファイル／ディレクトリの検査 (先の例の続き)

```
>>> os.path.isfile('file.txt') Enter ←存在するパス file.txt がファイルかどうかを検査
True ←真 (ファイルである)
>>> os.path.isdir('file.txt') Enter ←それがディレクトリかどうかを検査すると…
False ←偽 (ディレクトリではない)
>>> os.path.isdir('.') Enter ←カレントディレクトリ '.' がディレクトリかどうかを検査
True ←真 (ディレクトリである)
```

2.7.7.7 ファイル、ディレクトリの削除

ファイルを削除するには `remove` 関数を使用して、

`os.remove(削除対象のファイルのパス)`

とする。また空ディレクトリを削除するには `rmdir` 関数を使用して、

`os.rmdir(削除対象のディレクトリのパス)`

とする。削除対象のディレクトリの配下にはファイルやディレクトリがあってはならない。

`os` モジュールにはこの他にも様々なメソッドが用意されている。詳しくは巻末付録「A.1 Python のインターネットサイト」(p.400) を参照のこと。

2.7.7.8 実行中のスクリプトに関する情報

グローバル変数 `__file__` は、実行中のスクリプトのファイルのパスの文字列¹²² を保持している。この値を `os.path.abspath` 関数の引数に与えると、そのファイルの絶対パスの文字列が得られる¹²³。パスの文字列からディレクトリの部分のみを取り出すには `os.path.dirname` 関数を使用する。

これらを応用したプログラム `selfpath01.py` を示す。

プログラム：`selfpath01.py`

```
1 # coding: utf-8
2 # モジュールの読み込み
3 import os
4
5 print('グローバル変数__file__ :',__file__)
6
7 p = os.path.abspath( __file__ )
8 print('絶対パス :',p)
9
10 d = os.path.dirname( p )
11 print('ディレクトリ :',d)
```

このプログラムは、自身のファイル名、自身の絶対パス、自身のディレクトリを表示するものである。実行例を次に示す。

実行例.

```
C:\Users\katsu>py selfpath01.py Enter ←プログラムの実行開始
グローバル変数__file__ : C:\Users\katsu\selfpath01.py
絶対パス : C:\Users\katsu\selfpath01.py
ディレクトリ : C:\Users\katsu
```

2.7.7.9 パスの表現に関すること

ファイルのパスを表現する文字列を分解、あるいは合成する方法について説明する。

例. パス文字列の分解 (`os` モジュールは読み込み済みとする)

```
>>> p = '/home/katsu/python/prog1.py' Enter ←パス文字列の作成
>>> b = os.path.basename( p ) Enter ←右端の要素を取り出す
>>> b Enter ←内容確認
'prog1.py' ←ファイル名が得られた
>>> d = os.path.dirname( p ) Enter ←右端の要素を除外したものを取り出す
>>> d Enter ←内容確認
'/home/katsu/python' ←ディレクトリ名が得られた
```

この例では変数 `p` にパス文字列を作成し、それを `basename` 関数、`dirname` 関数で分解している。`basename` 関数はパス文字列の右端の要素を、`dirname` 関数は右端の要素を除外したものを返す。またこの例では、パス文字列中のディレクトリの区切りの文字としてスラッシュ「/」を用いているが、Windows 環境ではディレクトリの区切り文字は「¥」

¹²² 古い版の Python ではファイル名の文字列を保持している場合があるので注意されたい。

¹²³ この処理は特に `__file__` がファイル名のみ文字列を保持している場合に有用である。

が基本¹²⁴ である。

basename, dirname 関数と同様のことが os.path.split 関数でも可能である。

例. split 関数によるパス文字列の分解 (先の例の続き)

```
>>> os.path.split( p )  [Enter]    ←パス文字列の分解
(' /home/katsu/python', 'prog1.py')  ←戻り値
```

split 関数は分解結果をタプルにして返す。

ファイル名などの文字列が拡張子を持つ場合, splitext 関数で拡張子とそれ以外の部分に分解することができる。

例. 拡張子を持つ文字列の分解 (先の例の続き)

```
>>> os.path.splitext( b )  [Enter]    ←ファイル名の分解
('prog1', '.py')           ←分解結果のタプル
```

パス文字列を連結するには os.path.join 関数を使用する。ただしこの場合は、ディレクトリの区切り文字に注意する必要がある。

例. Windows 環境でのパス文字列の連結: その 1 (先の例の続き)

```
>>> os.path.join( d, b )  [Enter]    ←パス文字列を連結する試み
'/home/katsu/python¥¥prog1.py'      ←得られたパス文字列の区切り文字が適切でない
```

Windows 環境ではディレクトリの区切り文字が「¥」であるのでこのような結果となる。使用しているシステムでのディレクトリの区切り文字は os.sep を参照¹²⁵ して確認できる。

例. Windows 環境でのディレクトリの区切り文字の確認 (先の例の続き)

```
>>> os.sep  [Enter]    ←確認
'¥¥¥'       ←「¥」が区切り文字であることがわかる
```

Windows 環境での正しい実行例を次に示す。

例. 'C:¥Users¥katsu' と 'programing¥python' を連結する

```
>>> p = os.path.join( 'C:¥¥Users¥¥katsu', 'programing¥¥python' )  [Enter]    ←パスの連結
>>> print( p )  [Enter]    ←内容確認
C:¥Users¥katsu¥programing¥python      ←結果表示
```

■ 絶対パス／相対パスの判定

表現されたパスが絶対パスであるかどうかを判定するには isabs 関数を使用する。

例. 絶対パス／相対パスの判定 (先の例の続き)

```
>>> p1 = '/etc/passwd'; p2 = 'taro/prg1.py'  [Enter]    ←パスの文字列 2 種
>>> os.path.isabs(p1)  [Enter]    ← p1 は
True                  ←絶対パスである
>>> os.path.isabs(p2)  [Enter]    ← p2 は
False                 ←絶対パスではない
```

Windows のパスの場合、パスの左端に「C:¥」もしくは「¥」があると True となる。

この他にも os モジュールには多くの機能が提供されている。更に詳しくは Python の公式インターネットサイトを参照されたい。

2.7.8 パス (ファイル, ディレクトリ) の扱い: その 2 - pathlib モジュール

Python 3.4 から標準ライブラリとして導入された pathlib は、ファイルとディレクトリを扱うための別の方法を提供する。このライブラリを使用するには次のようにして読み込む。

```
from pathlib import Path
```

¹²⁴Windows 環境でも多くの場合、パス文字列中にスラッシュ「/」が使える。

¹²⁵os.path.sep も同様。

2.7.8.1 パスオブジェクトの生成

pathlib では、パスを文字列 (str オブジェクト) としてではなく、Path クラスのオブジェクトとして扱う。Path オブジェクトを生成するにはコンストラクタの引数にはパスの文字列を与える。

例. /Users/katsu/test01.txt を表すパスオブジェクト

```
p = Path('/Users/katsu/test01.txt')
```

この例はファイルへのパス /Users/katsu/test01.txt を表す Path オブジェクトを p として生成するものである。

2.7.8.2 カレントディレクトリ、ホームディレクトリの取得

カレントディレクトリ、ホームディレクトリのパスを取得する方法について、Windows 環境における実行例を示す。

例. カレントディレクトリを取得する Path.cwd() メソッド

```
>>> Path.cwd() Enter ←カレントディレクトリの取得
WindowsPath('C:/Users/katsu/Python') ←カレントディレクトリ
```

例. ホームディレクトリを取得する Path.home() メソッド

```
>>> Path.home() Enter ←ホームディレクトリの取得
WindowsPath('C:/Users/katsu') ←ホームディレクトリ
```

2.7.8.3 パスの存在の検査

パスオブジェクトに対して exists メソッドを使用することで、そのパスが存在するかどうかを調べることができる。(exists の引数は空にする) そのパスが存在する場合は True を、存在しない場合は False を返す。

2.7.8.4 ファイル、ディレクトリの検査

パスオブジェクト p がファイルかディレクトリかを調べるには、例えば表 21 のようにして is_file, is_dir メソッドを使用する。

表 21: ファイル、ディレクトリの判定

p.is_file()	p がファイルの場合に True を、それ以外の場合に False を返す。
p.is_dir()	p がディレクトリの場合に True を、それ以外の場合に False を返す。

2.7.8.5 ディレクトリの要素を取得する

パスオブジェクト p がディレクトリの場合、glob メソッド¹²⁶ を使用して、配下の要素 (ファイル、サブディレクトリなど) を取得することができる。glob メソッドの引数にはワイルドカード (表 22) を含むパターンを文字列型で与える。

表 22: 重要なワイルドカード (一部)

記号	意味	記号	意味
*	任意の長さの任意の文字列	?	任意の 1 文字

例えば、

```
plst = list( p.glob('*') )
```

とすると、ディレクトリ p の配下の要素のリスト plst が得られる。また、

```
plst = list( p.glob('*.jpg') )
```

とすると、ファイル名の末尾が '.jpg' であるような要素をディレクトリ p の配下から探す。

glob メソッドの戻り値は Path オブジェクトの generator¹²⁷ であり、上の説明ではそれをリストに変換している。

2.7.8.6 ディレクトリ名、ファイル名、拡張子の取り出し

Path オブジェクトのプロパティには表 23 のようなものがある。

¹²⁶同様のメソッドが使える glob モジュールも存在するが、pathlib モジュールの方が便利である。

¹²⁷p.297「4.7 ジェネレータ」で説明する。

表 23: Path オブジェクトのプロパティ (一部)

プロパティ	説明
.name	Path オブジェクトの末尾の名前 (文字列型)
.suffix	Path オブジェクトの拡張子 (文字列型)
.parent	Path オブジェクトのディレクトリ部分 (Path オブジェクト)

例. Path オブジェクトのプロパティ (Windows での例)

```
>>> from pathlib import Path  Enter    ←モジュールの読み込み
>>> p = Path('C:\\Users\\katsu\\a.exe')  Enter    ← Path オブジェクトの生成
>>> p.name  Enter    ←ファイル名の取得
'a.exe'    ←ファイル名の部分
>>> p.suffix  Enter    ←拡張子の取得
'.exe'     ←拡張子の部分
>>> p.parent  Enter    ←上位ディレクトリ名の取得
WindowsPath('C:/Users/katsu')    ←上位ディレクトリ (Path オブジェクト)
```

2.7.8.7 パスの連結

パスオブジェクトに対して二項演算子 '/' を使用することで、パスの連結ができる。例えば、`p = Path('/Users')` に対して `p / 'katsu'` と記述すると、それは `'/Users/katsu'` を意味するパスオブジェクトとなる。

2.7.8.8 ファイルシステム毎のパスの表現

pathlib では基本的なパスのクラスとして Path を使用するが、Python 処理系を実行する OS によってパスのオブジェクトの表現が異なる。次に示すのは Apple 社の macOS における実行例である。

例. macOS 上の Python3 での Path.home() の実行

```
>>> Path.home()  Enter    ←ホームディレクトリの取得
PosixPath('/Users/katsu')    ←ホームディレクトリ
```

パスオブジェクトは Windows 環境下では WindowsPath クラス、macOS などの UNIX 系 OS の環境下では PosixPath クラスのオブジェクトとして扱われる。

2.7.8.9 URI への変換

パスオブジェクトに `as_uri` メソッドを実行すると URI (Uniform Resource Identifier)¹²⁸ 形式の文字列が得られる。

例. パスオブジェクトを示す URI の取得

```
>>> p = Path.home()  Enter    ← p にホームディレクトリのパスを取得
>>> p.as_uri()  Enter    ← p を URI に変換
'file:///C:/Users/katsu'    ←得られた URI
```

2.7.8.10 ファイルのオープン

パスオブジェクトに対して `open` メソッドを使用してファイルを開くことができる。このとき `open` メソッドの引数にモードやエンコーディングを与える。例えば、パスオブジェクト `p` をエンコーディングが utf-8 のテキスト形式として読取り用に開くには次のようにする。

```
f = p.open('r', encoding='utf-8')
```

処理の結果、ファイルオブジェクト `f` が返される。

¹²⁸RFC 3986

2.7.8.11 ファイル入出力

pathlib はファイル入出力のための簡便な方法を提供する。(pathlib はパスの扱いやファイル入出力に関する一連の機能を提供する)¹²⁹

テキストファイルのパスを表す Path オブジェクトに対して read_text メソッドを実行することで、そのファイルの内容を全て読み取って文字列として返す。

書き方： Path オブジェクト.read_text(encoding=エンコーディング)

この処理の前後にオープンやクローズの処理は必要ない。

例. Path オブジェクトが示すテキストファイルから内容を読み込む

```
>>> p = Path('./dat1.txt')  Enter    ←テキストファイルのパス
>>> txt = p.read_text( encoding='utf-8' )  Enter    ←内容の読み込み
>>> print( txt )  Enter    ←内容確認
1 行目    ←内容表示
2 行目
3 行目
```

この例では、カレントディレクトリにあるテキストファイル 'dat1.txt' の内容を全て読み取り、それを txt に与えている。

Path オブジェクトに対して write_text メソッドを実行することで、それが示すファイルに文字列を出力することができる。

書き方： Path オブジェクト.write_text(文字列, encoding=エンコーディング)

この処理の前後にオープンやクローズの処理は必要ない。処理の後、出力した文字数を返す。

例. Path オブジェクトが示すファイルに文字列を出力する（先の例の続き）

```
>>> p2 = Path('./dat2.txt')  Enter    ←出力先のパス
>>> p2.write_text( txt, encoding='utf-8' )  Enter    ←文字列の出力
12      ←出力した文字数が返される
```

この例では、文字列 txt をファイル 'dat2.txt' に出力している。

ファイルのパスを表す Path オブジェクトに対して read_bytes メソッドを実行することで、そのファイルの内容を全て読み取ってバイト列として返す。

書き方： Path オブジェクト.read_bytes()

この処理の前後にオープンやクローズの処理は必要ない。これはバイナリデータをファイルから読み込む手段となる。

例. Path オブジェクトが示すファイルから内容をバイナリデータとして読み込む（先の例の続き）

```
>>> buf = p.read_bytes()  Enter    ←内容の読み込み（バイナリデータ）
>>> txt2 = buf.decode('utf-8')  Enter    ←それをテキスト形式（文字列）に変換
>>> print( txt2 )  Enter    ←内容確認
1 行目    ←内容表示
2 行目
3 行目
```

この例では、先の例で作成した Path オブジェクトからバイナリデータとして内容を読み込んでいる。読み込んだ内容はバイト列として buf に得られている。

Path オブジェクトに対して write_bytes メソッドを実行することで、それが示すファイルにバイト列を出力することができる。

書き方： Path オブジェクト.write_bytes(バイト列)

この処理の前後にオープンやクローズの処理は必要ない。処理の後、出力したバイト数を返す。これはバイナリデータをファイルに出力する手段となる。

¹²⁹ここで説明する機能は Python3.5 以降で有効である。

例. Path オブジェクトが示すファイルにバイト列を出力する（先の例の続き）

```
>>> p3 = Path('./dat3.txt')  Enter    ←出力先のパス
>>> p3.write_bytes( buf )  Enter    ←バイト列の出力
27                            ←出力したバイト数が返される
```

この例では、先の例で得たバイト列 buf をファイル 'dat3.txt' に出力している。

2.7.8.12 ディレクトリの作成

Path オブジェクトに対して mkdir メソッドを実行することで、それが示すディレクトリを作成することができる。

書き方：Path オブジェクト.mkdir()

例. ディレクトリの作成（先の例の続き）

```
>>> p4 = Path('./dir01')  Enter    ←作成するディレクトリのパス
>>> p4.mkdir()  Enter    ←ディレクトリの作成
```

この例では、ディレクトリ 'dir01' を作成している。

2.7.8.13 ファイル、ディレクトリの削除

Path オブジェクトに対して unlink メソッドを実行すると、それが示すファイルを削除することができる。

書き方：Path オブジェクト.unlink()

戻り値は無い。(None)

例. ファイルの削除（先の例の続き）

```
>>> p2.unlink()  Enter    ← Path オブジェクト p2 が示すファイルを削除
>>> p3.unlink()  Enter    ← Path オブジェクト p3 が示すファイルを削除
```

Path オブジェクトに対して rmdir メソッドを実行すると、それが示すディレクトリを削除することができる。

書き方：Path オブジェクト.rmdir()

戻り値は無い。(None)

例. ディレクトリの削除（先の例の続き）

```
>>> p4.rmdir()  Enter    ← Path オブジェクト p4 が示すディレクトリを削除
```

2.7.8.14 Path オブジェクトの文字列への変換

Path オブジェクトは str 関数で文字列に変換すると、当該処理環境のパス表記の文字列が得られる。(次の例)

例. Path オブジェクトを文字列に変換する例（Windows 環境）

```
>>> from pathlib import Path  Enter    ←モジュールの読み込み
>>> p = Path('./a/b/c')  Enter    ← Path オブジェクトの作成（相対パス）
>>> p  Enter    ←内容確認
WindowsPath('a/b/c')    ← Windows 用の Path オブジェクトができています
>>> str(p)  Enter    ←文字列に変換
'a\\b\\c'              ← Windows 用のパスの表記になっている
```

例. 絶対パスの場合（先の例の続き）

```
>>> p = Path('c:/a/b/c')  Enter    ← Path オブジェクトの作成（絶対パス）
>>> p  Enter    ←内容確認
WindowsPath('c:/a/b/c')    ← Windows 用の Path オブジェクト
>>> str(p)  Enter    ←文字列に変換
'c:\\a\\b\\c'          ← Windows 用のパスの表記になっている
```

付録「I.4 pathlib の応用例」(p.461) に pathlib を応用したサンプルプログラムを掲載する。

2.7.9 コマンド引数の取得

ソースプログラムを Python 処理系（インタプリタ）にスクリプトとして与えて実行を開始する際、起動時に与えたコマンド引数を取得するには `sys` モジュールのプロパティ `argv` を参照する。次のプログラム `test17.py` の実行を例にして説明する。

プログラム：test17.py

```
1 # coding: utf-8
2 # 必要なモジュールの読み込み
3 import sys
4
5 # コマンド引数の取得
6 print('args>', sys.argv)
```

このプログラムは、起動時のコマンド引数の列をリストにして表示するものであり、実行すると次のように表示される。

```
C:\Users\katsu> py test17.py 1 2 3 a b  ← OS のコマンドラインから起動（Windows の場合）
args> ['test17.py', '1', '2', '3', 'a', 'b'] ←与えた引数が得られる
```

このように、ソースプログラム名から始まる引数が文字列のリストとして得られることがわかる。コマンド引数からプログラムに数値を与える場合は、文字列として得られた引数を、`int` 関数 や `float` 関数を用いて適切な型に変換すると良い。

Python で実用的なコマンドツールを作成する際には、コマンド起動時に与えられた引数を解析するための更に高度な機能が求められることがある。それに関しては後の「4.28 コマンド引数の扱い：argparse モジュール」（p.356）で解説する。

2.7.10 入出力処理の際に注意すること

実際のシステムにおける入出力処理の際には、エラー（例外）が発生することがある。例えば、ファイルのオープンや読み込み、書き出しの際に、その処理が実行できない状況が発生しうる。具体的な例外事象としては、存在しないファイルを読み込み用にオープンしようとしたり、アクセス権限の無いシステム資源に対して入出力を試みたりと枚挙にいとまがない。従って、実用的なアプリケーションプログラムを作るにあたっては、発生しうる例外事象を十分に想定して対応策の処理¹³⁰ を行う形で実装しなければならない。

Python では、`with` 構文を用いた記述が可能であり、ファイルのオープンとクローズや例外処理のハンドリングを簡潔に記述することができる。これに関しては「4.21 `with` 構文」（p.340）で説明する。

2.7.11 CSV ファイルの取り扱い：csv モジュール

表形式のデータを表現するものに **CSV 形式** のデータフォーマット¹³¹ があり、多くのアプリケーションソフトウェア¹³² においてこの形式のデータを取り扱うことができ、異なるアプリケーション間での表形式データの交換に広く用いられている。

CSV 形式では値をコンマ「`,`」で区切って表現し、そのようなデータ並びで**行**（レコード）を構成する。また、CSV 形式のデータファイルはそのような複数の行から構成される。

Python 言語ではリストに対する `join` メソッドを応用することで CSV 形式のデータ（文字列）を構成することができ、逆に、文字列に対する `split` メソッドを応用することで CSV の各値を分離することができるが、ファイル入出力の目的で CSV データをより簡便な形で取り扱うために **csv モジュール** が標準的に提供されており、これを利用することができる。csv モジュールを利用するには

```
import csv
```

として Python 処理系に読み込み、「`csv.`」の接頭辞を付けて各種の機能呼び出す。

¹³⁰ 「2.5.1.6 例外処理」（p.63）、「4.19 例外（エラー）の処理」（p.337）を参照のこと。

¹³¹ RFC4180 として標準化されている。

¹³² Microsoft 社の Excel などが有名。

2.7.11.1 CSV ファイルの出力

ここでは、CSV データのファイルへの出力に関して例を挙げて説明する。まず次のような処理によってサンプルデータをリストの形で作成する。

例. サンプルデータ（リスト）の作成

```
>>> tbl = [[x+y for y in range(1,5)] for x in range(10,40,10)] Enter ←サンプルデータの作成
>>> for r in tbl: print( r ) Enter ←内容確認（繰り返し処理）
... Enter ←繰り返し処理の記述の終了
[11, 12, 13, 14] ← 3 行 4 列の表がリストの形式で tbl に得られている
[21, 22, 23, 24]
[31, 32, 33, 34]
```

この例で得られたデータ tbl を CSV 形式ファイルに出力するには for などによる反復制御で tbl の各要素を出力しても良いが、csv モジュールを利用することでその処理が簡略化される。

【CSV ファイルの出力の手順】

1. open 関数で出力対象ファイルのファイルオブジェクトを作成する。
2. 上で作成したファイルオブジェクトから **writer オブジェクト** を作成する。
3. 上で作成した writer オブジェクトに対して **writerow** メソッドもしくは **writerows** メソッドを用いて CSV データを出力する。
4. 出力処理が終われば、1 で作成したファイルオブジェクトに対して **close** メソッドを実行してファイルを閉じる。

この手順に従って、先に作成したリスト tbl の内容を出力する例を示す。

例. writerow による CSV ファイルの出力（先の例の続き）

```
>>> import csv  [Enter]    ←モジュールの読み込み
>>> f = open('csvTest01.csv','w',newline='') [Enter]    ←出力ファイルのオープン133
>>> cw = csv.writer(f) [Enter]    ←上記 f から writer オブジェクトを作成
>>> for r in tbl: [Enter]    ←繰り返し処理によって
...     b = cw.writerow(r) [Enter]    ← CSV を 1 行ずつ出力する
... [Enter]    ←繰り返しの記述の終了
>>> f.close() [Enter]    ←出力ファイルのクローズ
```

この例では writerow メソッドで CSV データを 1 行ずつ出力している。

書き方： writer オブジェクト.writerow(リスト)

1 次元の「リスト」の内容を「writer オブジェクト」を通して CSV 形式で 1 行出力する。writerow メソッドの内部では、ファイルオブジェクトに対する write メソッドの処理を応用しており、内部で実行した write メソッドの戻り値を結果として返す。

例に示した処理によって次のような CSV ファイル csvTest01.csv が作成される。

CSV ファイル：csvTest01.csv

```
1 11,12,13,14
2 21,22,23,24
3 31,32,33,34
```

上の例と同様の処理は writerows メソッドを用いることで更に簡潔に記述できる。

例. writerows による CSV ファイルの出力（先の例の続き）

```
>>> f = open('csvTest01.csv','w',newline='') [Enter]    ←出力ファイルのオープン
>>> cw = csv.writer(f) [Enter]    ← writer オブジェクトの作成
>>> cw.writerows(tbl) [Enter]    ← tbl の内容を一度に出力
>>> f.close() [Enter]    ←出力ファイルのクローズ
```

この例では writerows メソッドで CSV データを複数行まとめて出力している。

書き方： writer オブジェクト.writerows(リスト)

入れ子になった 2 次元の「リスト」の内容を「writer オブジェクト」を通して CSV 形式で出力する。すなわち「リスト」の各要素を各行とする CSV データとして出力する。

参考) writerow, writerows にはリスト以外のイテラブルなオブジェクトを引数に与えることもできる。

■ 区切り文字の指定方法

CSV 形式はコンマで区切られたものであるが、別の文字を区切り文字に用いることもできる。具体的には writer オブジェクト生成時にキーワード引数 'delimiter=区切り文字' を与える。

例. 出力ファイルの区切り文字をタブ「¥t」にする（先の例の続き）

```
>>> f = open('csvTest02.tsv','w',newline='') [Enter]    ←出力ファイルのオープン
>>> cw = csv.writer(f,delimiter='¥t') [Enter]    ← writer オブジェクト作成：区切り文字を指定
>>> cw.writerows(tbl) [Enter]    ← tbl の内容を一度に出力
>>> f.close() [Enter]    ←出力ファイルのクローズ
```

この処理によって次のような CSV ファイル csvTest02.tsv が作成される。

CSV ファイル：csvTest02.tsv

```
1 11 12 13 14
2 21 22 23 24
3 31 32 33 34
```

¹³³このように「newline=」として改行コードを指定すると安全である。

■ 辞書オブジェクトを CSV ファイルに出力する方法

DictWriter オブジェクトを用いることで、辞書オブジェクトを CSV データとして出力することができる。次のようにして作成した辞書オブジェクトを CSV データとして出力する例を示す。

例. サンプルの辞書オブジェクトの作成（先の例の続き）

```
>>> d = { 'col1':1, 'col2':2, 'col3':3, 'col4':4, 'col5':5 } Enter ←辞書オブジェクト作成
>>> d Enter ←内容確認
{'col1': 1, 'col2': 2, 'col3': 3, 'col4': 4, 'col5': 5}
```

DictWriter による出力では、出力対象の CSV ファイルの各列は**フィールド名**（カラム名）を持つ。そして、出力する辞書オブジェクトの各キーを CSV のフィールドに対応させる。次に、実際に辞書オブジェクトを CSV ファイルに出力（1 行のみ）する例を示す。

例. 辞書オブジェクトを CSV データとして出力する（先の例の続き）

```
>>> f = open('csvTest03.csv','w',newline='') Enter ←出力ファイルのオープン
>>> cn = ['col5','col3','col1'] Enter ← CSV 用の見出し（フィールド）の定義
>>> dw = csv.DictWriter( f, fieldnames=cn, extrasaction='ignore' ) Enter ← DictWriter 作成
>>> dw.writeheader() Enter ←見出し行の出力
16 Enter ←上記処理の戻り値
>>> b = dw.writerow( d ) Enter ←辞書 d を CSV データの 1 つの行として出力
>>> f.close() Enter ←出力ファイルのクローズ
```

辞書オブジェクト d は 'col1'～'col5' の 5 つのエントリを持つが、この例では 'col1', 'col3', 'col5' の 3 つのエントリを出力対象としており、そのリストを変数 cn に与えている。この出力対象のフィールドのリストは DictWriter オブジェクト生成時にキーワード引数

fieldnames=出力するフィールドのリスト

として与える。またこの例のように、辞書オブジェクトが出力対象としないエントリを含んでいる場合はキーワード引数「extrasaction='ignore'」を与える¹³⁴。

CSV 形式ファイルは先頭の行がフィールド名になっていることが一般的である。上の例の writeheader メソッドは出力先の CSV ファイルにフィールド名の見出しを出力するものである。

上の例の処理によって次のような CSV ファイル csvTest03.csv が作成される。

CSV ファイル：csvTest03.csv

```
1 col5,col3,col1
2 5,3,1
```

DictWriter オブジェクトに対して writerows メソッドを用いると、複数の辞書を連続して出力することができる。そのことをプログラム Dic2Csv01.py で示す。

プログラム：Dic2Csv01.py

```
1 # coding: utf-8
2 import csv
3 # 出力用データ（3レコード分の辞書をリスト形式で用意）
4 dLst = [ { 'A':11, 'B':12, 'C':13, 'D':14, 'E':15 },           # 辞書1
5          { 'A':21, 'B':22, 'C':23, 'D':24, 'E':25 },           # 辞書2
6          { 'A':31, 'B':32, 'C':33, 'D':34, 'E':35 } ]          # 辞書3
7 # 出力処理
8 f = open('csvTest03-2.csv','w',newline='')                    # 出力ファイルをオープン
9 dw = csv.DictWriter(f,fieldnames=['A','C','E'],extrasaction='ignore')
10 dw.writeheader()      # 見出し行の出力
11 dw.writerows(dLst)    # リストの各要素（辞書）を出力
12 f.close()             # 出力ファイルのクローズ
```

¹³⁴DictWriter による辞書オブジェクトの出力では、辞書オブジェクトの全てのエントリのキーが出力先 CSV のフィールドに対応しなければならない。この制限を解除するためにこのキーワード引数 extrasaction を与える。

解説)

このプログラムでは、出力する複数のレコードとなる辞書をリスト `dLst` として用意（4～6 行目）している。そして `dLst` の全ての要素について、キー 'A', 'C', 'E' のエントリを抽出して `writerows` メソッドで出力（11 行目）している。

このプログラムを実行すると、ファイル `csvTest03-2.csv` のような CSV データが出来上がる。

CSV ファイル：`csvTest03-2.csv`

1	A,C,E
2	11,13,15
3	21,23,25
4	31,33,35

2.7.11.2 CSV ファイルの入力

CSV 形式のテキストファイルを入力する方法について解説する。

【CSV ファイルの入力の手順】

1. `open` 関数で入力ファイルのファイルオブジェクトを作成する。
2. 上で作成したファイルオブジェクトから **reader オブジェクト** を作成する。
3. 上で作成した reader オブジェクトをイテラブルとみなしてデータを順次取り出す。
4. 入力処理が終われば、1 で作成したファイルオブジェクトに対して `close` メソッドを実行してファイルを閉じる。

先の例で作成した CSV ファイル `csvTest01.csv` (p.130) を読み込む処理を例に挙げて、CSV ファイルの入力について解説する。

例. CSV ファイルの読み込み（先の例の続き）

```
>>> f = open('csvTest01.csv','r')  [Enter]    ← CSV ファイルのオープン
>>> cr = csv.reader(f)  [Enter]    ← reader オブジェクト作成
>>> for r in cr:  [Enter]    ← reader オブジェクトからデータを取り出す反復制御
...     print(r)  [Enter]    ← 入力した行を表示
...  [Enter]    ← 反復制御の記述の終了
['11', '12', '13', '14']    ← 入力したデータ（リスト）
['21', '22', '23', '24']
['31', '32', '33', '34']
>>> f.close()  [Enter]    ← 入力ファイルのクローズ
```

この例では、入力用の CSV ファイルを開き、それを元に reader オブジェクト `cr` を作成し、`cr` からデータを 1 行ずつ読み込んでいる。このとき各行はリストの形で得られる。reader オブジェクトはイテラブルであるので様々な形で応用することができる。

例. CSV ファイルの内容を全て読み込む（先の例の続き）

```
>>> f = open('csvTest01.csv','r')  [Enter]    ← CSV ファイルのオープン
>>> cr = csv.reader(f)  [Enter]    ← reader オブジェクト作成
>>> tbl = [r for r in cr]  [Enter]    ← リストの内包表記によって全ての内容を読み込む
>>> f.close()  [Enter]    ← 入力ファイルのクローズ
>>> tbl  [Enter]    ← 内容確認
[['11', '12', '13', '14'],    ← 得られたデータ
 ['21', '22', '23', '24'],
 ['31', '32', '33', '34']]
```

この例ではリストの内包表記 `tbl = [r for r in cr]` を用いて CSV ファイルの全内容を読み込んでいるが、reader オブジェクト `cr` はイテラブルであるため `list(cr)` として全てを一度にリストに変換することもできる。

■ 見出し行をスキップする方法

CSV ファイルの先頭行が見出し行である場合、先頭行がデータでないことからそれを排除する場合がある。先頭行

を排除する最も単純な方法として、CSV から得られたリストの先頭要素を排除する方法がある。例えば、上の例で得られたリスト `tbl` の先頭要素を排除するには次のようにする。

例. 見出し行の排除（先の例の続き）

```
>>> tbl[1:]  ←先頭行の排除
[['21', '22', '23', '24'], ←先頭要素が排除されている
 ['31', '32', '33', '34']]
```

根本的な方法としては、`reader` オブジェクトから読み込む時点で最初の行をスキップしておくという方法もある。（次の例）

例. 見出し行の排除：その2（先の例の続き）

```
>>> f = open('csvTest01.csv', 'r')  ← CSV ファイルのオープン
>>> cr = csv.reader(f)  ← reader オブジェクト作成
>>> r = next(cr)  ← cr の最初の要素を読み飛ばす
>>> tbl = list(cr)  ← CSV ファイルの全ての内容を読み込む
>>> f.close()  ←入力ファイルのクローズ
>>> tbl  ←内容確認
[['21', '22', '23', '24'], ←先頭要素が排除されている
 ['31', '32', '33', '34']]
```

この例では、CSV データをリスト `tbl` に取得する前に `next` 関数で最初の要素（行）を読み飛ばしている。

■ CSV データの値を数値として読み込む方法

`reader` オブジェクトから読み込む値は文字列である。これを数値（整数、浮動小数点数）として読み込むには読み込み時に型を変換する。例えば先の例で使用した CSV ファイルの第 1 列を整数として、第 2 列を浮動小数点数として読み込むには次のような方法がある。

例. 型を指定して読み込む（先の例の続き）

```
>>> f = open('csvTest01.csv', 'r') 
>>> cr = csv.reader(f) 
>>> tbl = [ [int(r[0]),float(r[1]))+r[2:] for r in cr]  ←型を変換しながらリストに変換
>>> f.close()  ←入力ファイルのクローズ
>>> tbl  ←内容確認
[[11, 12.0, '13', '14'], ←第 1 列が整数, 第 2 列が浮動小数点数,
 [21, 22.0, '23', '24'], ←それ以降の列は文字列として得られている
 [31, 32.0, '33', '34']]
```

注意)

この方法では、CSV ファイルの中に数値に変換できないものが値として含まれていると `ValueError` が発生する。例えば次のような CSV ファイル `csvTest01-err.csv` を読み込む場合について考える。

CSV ファイル：`csvTest01-err.csv`

1	11,12,13,14
2	xx,22,23,24
3	31,yy,33,34

このファイルの 2 行目、3 行目に数値でないものが含まれている。先の例と同様の方法でこのファイルを読み込む試みを次に示す。

例. 数値に変換できない値を数値として読み込む試み（先の例の続き）

```
>>> f = open('csvTest01-err.csv','r') Enter
>>> cr = csv.reader(f) Enter
>>> tbl = [ [int(r[0]),float(r[1]))+r[2:] for r in cr] Enter ←数値として読み込もうとすると…
Traceback (most recent call last):      ←エラーが発生する
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <listcomp>
ValueError: invalid literal for int() with base 10: 'xx'      ← ValueError
```

これは、整数に変換しようとする値として文字列 'xx' が存在していることに起因するエラーである。このような場合、数値として解釈できない値を nan（非数）として扱うなどの工夫が必要となる。次にそのための工夫の1例を示す。

まず、文字列を float に変換するための関数 toFloat を次のように定義する。

例. 文字列を float 型に変換する関数の定義（先の例の続き）

```
>>> import math Enter      ← nan を使用するために math ライブラリを読み込む
>>> def toFloat(s): Enter    ←関数定義135 の開始
...     try: Enter
...         return float(s) Enter      ←文字列を float に変換したものを返す
...     except: Enter
...         return math.nan Enter      ←エラーが発生した場合は nan を返す
... Enter      ←関す定義の記述の終了
>>>      ←対話モードのプロンプトに戻った
```

これは「try…except～」の構文¹³⁶を応用して、float に変換できないものを float に変換しようとした場合に発生するエラーを受けて nan を返す関数である。この関数 toFloat を用いた例を次に示す。

例. 数値に変換できない値を nan に変換して読み込む（先の例の続き）

```
>>> f = open('csvTest01-err.csv','r') Enter
>>> cr = csv.reader(f) Enter
>>> tbl = [ [toFloat(r[0]),toFloat(r[1]))+r[2:] for r in cr] Enter ← toFloat で変換
>>> f.close() Enter
>>> tbl Enter      ←内容確認
[[11.0, 12.0, '13', '14'],
 [nan, 22.0, '23', '24'],      ←'xx' が nan に変換されている
 [31.0, nan, '33', '34']]      ←'yy' が nan に変換されている
```

読み込んだデータの第1列と第2列が float の型に変換されている。nan（非数）は float 型であるので、この例では第1列、第2列共に float に変換することとした。

■ 区切り文字の指定方法

CSV ファイルの区切り文字がコンマ以外のものである場合、reader オブジェクト作成時にキーワード引数 'delimiter=' 区切り文字' を与える。

例. 区切り文字を指定して入力する（先の例の続き）

```
>>> f = open('csvTest02.tsv','r') Enter      ← CSV ファイルのオープン
>>> cr = csv.reader(f,delimiter='\\t') Enter  ← reader オブジェクト作成：区切り文字指定
>>> tbl = [r for r in cr] Enter      ←リストの内包表記によって全ての内容を読み込む
>>> f.close() Enter      ←入力ファイルのクローズ
>>> tbl Enter      ←内容確認
[['11', '12', '13', '14'],      ←得られたデータ
 ['21', '22', '23', '24'],
 ['31', '32', '33', '34']]
```

¹³⁵def 文による関数の定義に関しては「2.8 関数の定義」(p.137) で解説する。

¹³⁶詳しくは「2.5.1.6 例外処理」(p.63)、「4.19 例外（エラー）の処理」(p.337) を参照のこと。

これは p.130 で作成したタブ区切りの CSV ファイル csvTest02.tsv を読み込む処理である。

■ CSV ファイルからの入力を辞書オブジェクトにする方法

DictReader オブジェクトを用いることで、CSV ファイルから読み取った行を辞書オブジェクトにすることができる。

例. CSV ファイルの各行を辞書オブジェクトとして得る（先の例の続き）

```
>>> f = open('csvTest03.csv','r')  Enter    ← CSV ファイルのオープン
>>> dr = csv.DictReader( f )  Enter    ← DictReader オブジェクト作成
>>> d = [dict(r) for r in dr]  Enter    ← 読み取った各行を辞書にして、それらのリストを取得
>>> f.close()  Enter    ← 入力ファイルのクローズ
>>> d  Enter    ← 内容確認
[{'col5': '5', 'col3': '3', 'col1': '1'}]
```

この例では p.131 で作成した CSV ファイル csvTest03.csv の内容を読み取って、その行を辞書オブジェクトにしている。また、読み取った全内容はそれら辞書オブジェクトのリストとして得ている。得られた辞書オブジェクトのキーは CSV ファイルの先頭行（見出し行）の項目から得ている。またこの例ではリストの内包表記を応用して `d = [dict(r) for r in dr]` として CSV ファイルの全内容を読み取っているが、DictReader オブジェクト `dr` はイテラブルなので `list(dr)` として全内容を読み取ることもできる。

CSV ファイルの全ての行（先頭行も含む）をデータとみなして読み込むには DictReader オブジェクト作成時にキーワード引数

fieldnames=フィールド名のリスト

を与える。これにより「フィールド名のリスト」の要素をキーとする辞書オブジェクトが得られる。

例. 全ての行をデータとみなして辞書オブジェクトの形で読み込む（先の例の続き）

```
>>> f = open('csvTest03.csv','r')  Enter    ← CSV ファイルのオープン
>>> cn = ['A','B','C']  Enter    ← フィールド名のリスト
>>> dr = csv.DictReader( f, fieldnames=cn )  Enter    ← DictReader 作成：フィールド名指定
>>> d = list(dr)  Enter    ← 読み取った各行を辞書にして、それらのリストを取得
>>> f.close()  Enter    ← 入力ファイルのクローズ
>>> d  Enter    ← 内容確認
[ {'A': 'col5', 'B': 'col3', 'C': 'col1'},    ← 先頭の行もデータとして読み込んでいる
  {'A': '5', 'B': '3', 'C': '1'} ]          ← 次の行のデータ
```

この例では入力用 CSV ファイルの各列のフィールド名を 'A', 'B', 'C' と見なしている。それらのリストを変数 `cn` に与えて、それを DictReader のキーワード引数 `fieldnames=` に与えている。これによって CSV ファイルの先頭行もデータとして読み込まれている。

■ クォート文字を含む CSV ファイルの読み込み

CSV 形式ファイルの中の値（項目）はクォート文字を含むものがある。例えば、コンマ文字を含む項目を 1 つの値として取り扱う場合は、データ項目の区切り文字としてのコンマ文字と区別する必要がある。例えば表計算ソフトウェアで図 7 のような表を作成した状況を考える。

	A	B	C
1	番号	語	意味
2	1	dog	犬
3	2	cat	猫
4	3	knife,blade	ナイフ

図 7: Microsoft Excel で CSV ファイルを作成する

この表は 1 つのセル（B4）の内容がコンマ文字を含んでおり、これを CSV ファイルとして保存¹³⁷ すると次に示すファイル csvExcel01.csv のようになる。

¹³⁷ 保存時のファイルの種類として「CSV UTF-8 (カンマ区切り) (*.csv)」を選択する。


```

番号, 語, 意味
1,dog, 犬
2,cat, 猫
3,"knife,blade", ナイフ

```

このファイルの4行目の2番目の項目を見ると、データとしてのコンマ文字を区切り文字と区別するために二重引用符「"」で括られているのがわかる。このファイルを DictReader オブジェクトで読み込む例を示す。

例. コンマ文字を含む項目の取り扱い

```

>>> import csv  Enter  ←モジュールの読み込み
>>> f = open('csvExcel01.csv','r',encoding='utf-8-sig')  Enter  ← CSV ファイルのオープン138
>>> dr = csv.DictReader( f )  Enter  ← DictReader 作成
>>> d = [dict(r) for r in dr]  Enter  ←読み取った各行を辞書にして、それらのリストを取得
>>> f.close()  Enter  ←入力ファイルのクローズ
>>> d  Enter  ←内容確認
[ {'番号': '1', '語': 'dog', '意味': '犬'},          ←読み込んだ内容
  {'番号': '2', '語': 'cat', '意味': '猫'},
  {'番号': '3', '語': 'knife,blade', '意味': 'ナイフ'} ]

```

データとしてのコンマ文字がデータとして読み込まれていることがわかる。これに対して、このファイル csvExcel01.csv を通常のテキストファイルのように読み込み、改行コードとコンマ文字で split すると問題が起こる。(次の例参照)

例. コンマ文字を含む項目を正しく取り扱えないケース (先の例の続き)

```

>>> f = open('csvExcel01.csv','r',encoding='utf-8-sig')  Enter  ← CSV ファイルのオープン
>>> lines = f.read().splitlines()  Enter  ←ファイルの全内容を読み込み行を分割
>>> tbl = [lin.split(',') for lin in lines]  Enter  ←各行をコンマ文字で分割
>>> f.close()  Enter  ←入力ファイルのクローズ
>>> tbl  Enter  ←内容確認
[ ['番号', '語', '意味'],
  ['1', 'dog', '犬'],
  ['2', 'cat', '猫'],
  ['3', '"knife', 'blade"', 'ナイフ']]  ←コンマ文字の取り扱いが正しくない

```

データの最後の行の中の項目 'knife,blade' のコンマ文字が区切り文字として認識され、その項目が正しくない形で分割されている。この例からもわかるように、CSV ファイルの扱いには csv モジュールを使用すべきであることがわかる。

参考) 配列処理用のライブラリ NumPy や、データ処理用ライブラリ pandas などを用いると更に簡便な方法で CSV 形式ファイルを取り扱うことができる¹³⁹。

¹³⁸Microsoft Excel の表を「BOM 付き UTF-8」のエンコーディングで保存した CSV ファイルを読み込む際はこのようにしてオープンする。

¹³⁹NumPy については拙書「Python3 ライブラリブック」で、pandas については拙書「Python3 によるデータ処理の基礎」で解説しています。

2.8 関数の定義

関数とは、

関数名 (引数列)

の形式で呼び出すサブプログラムで、メインプログラムや他のサブプログラムから呼び出す形で実行する。関数は、処理結果を何らかのデータとして返す(戻り値を持つ)ものである。Python 言語処理系は、予め多くの組込み関数を提供している¹⁴⁰ が、利用者が独自に関数を定義¹⁴¹ して使用することもできる。

《 関数定義の記述 》

書き方: `def 関数名 (仮引数列):`
 (処理内容)
 `return 戻り値`

「処理内容」から `return` までの行は、`def` よりも右の位置に同一の深さのインデント (字下げ) を施したスイートとして記述する。関数の処理は `return` のところで終了し、呼び出し元に実行の制御が戻る。処理の結果の戻り値を `return` の右側に記述するとその値が呼び出し元に返される。`return` の右側の戻り値は省略することができ、また `return` 文を省略してスイートの記述を終えることもできる。その場合は戻り値は `None` となる。

関数定義の例. 加算する関数 `kasan` の定義 (プログラム `test08-1.py`)

プログラム: `test08-1.py`

```
1 # 加算する関数
2 def kasan(x,y):
3     z = x + y
4     return z
5 # メインルーチン
6 a = kasan(12,34)
7 print(a)
```

このスクリプトファイルを Python 処理系が読み込むと内容が上から順番に解釈されるが、この時 2~4 行目の関数は「定義の記述」として扱われ、読み込みの段階ではまだ実行されない。そして処理系が 6 行目を読み込んだ時点で関数 `kasan` が実際に実行され、その戻り値 (`return` による計算結果) が変数 `a` に割当てられ、7 行目の `print` によってその値が出力される。実行結果として

46

と表示される。

■ 実引数, 仮引数

引数は関数名の後ろに括弧 '`()`' で括って記述する。関数の定義を記述する際に書き並べる引数を仮引数と呼び、その関数を呼び出す際に与える引数を実引数と呼ぶ。(図 8)

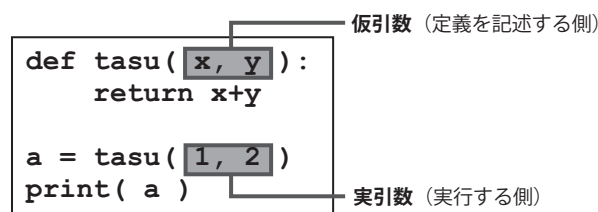


図 8: 引数の名称

2.8.1 引数について

関数はそれを呼び出したプログラムから仮引数に値を受け取って処理を行う。先のプログラム `test08-1.py` では関数 `kasan` は 2 つの仮引数を持ち、それらに受け取った値の加算を行っている。呼び出す側の実引数と関数定義に記述された仮引数は、記述された順序でそれぞれ対応する。この形式の引数を位置引数という。位置引数以外にも引数の受け渡しの形式があり、それらについては後に説明する。

¹⁴⁰ 全ての組込み関数の名前は `dir(__builtins__)` で得られるリストに含まれているので確認できる。

¹⁴¹ ユーザ定義関数と呼ぶことがある。

2.8.1.1 引数に暗黙値（デフォルト値）を設定する方法

関数の引数には暗黙値を設定することができる。（次の例）

例. 引数に暗黙値を取る関数

```
>>> def kasan2( x, y=1 ): Enter ←関数 kasan2 の定義の記述の開始
...     return x+y Enter
... Enter ←関数 kasan2 の定義の記述の終了
>>> kasan2( 1, 2 ) Enter ←引数を 2 つ与えて評価（実行）
3 ←評価結果
>>> kasan2( 1 ) Enter ← 2 つ目の引数を省略して評価（実行）
2 ←評価結果
```

この例では、関数 kasan2 の 2 番目の引数には暗黙値として 1 が設定されている。これにより、この関数の 2 番目の引数を省略して評価（実行）すると関数内では y=1 として扱われる。

■ 暗黙値を持つ引数の位置

暗黙値を持つ仮引数は、暗黙値を持たない仮引数の後にまとめて記述しなければならない。（次の例）

例. 暗黙値を持つ仮引数の正しい配置

```
>>> def f( a, b, c, d=1, e=2, f=3 ): pass Enter ←正しい配置
... Enter ←関数定義の記述の終了
```

この例では、暗黙値を持つ仮引数は、暗黙値を持たない仮引数の後にまとめて配置されている。

例. 暗黙値を持つ仮引数の正しくない配置

```
>>> def f( a, b=1, c, d=2, e, f=3 ): pass Enter ←正しくない配置
File "<stdin>", line 1
    def f( a, b=1, c, d=2, e, f=3 ): pass
    ^
SyntaxError: parameter without a default follows parameter with a default
```

この例では、暗黙値を持たない仮引数 c が、暗黙値を持つ仮引数 b の後に記述されており、文法エラー（SyntaxError）となる。

仮引数の暗黙値に関しては、後の「引数の暗黙値に関する事柄」（p.143）で更に解説する。

2.8.1.2 仮引数のシンボルを明に指定する関数呼び出し（キーワード引数）

関数を呼び出す際の実引数に、関数定義の記述にある仮引数の記号を明に指定することが可能である。例えば次のように定義された関数 fn があるとする。

例. サンプル用の関数 fn

```
>>> def fn(x,y): Enter
...     print(' 第 1 引数 x:',x) Enter
...     print(' 第 2 引数 y:',y) Enter
... Enter
>>> ← Python のプロンプトに戻った
```

この関数を呼び出す際に、仮引数の記号を使って fn(y=2,x=1) などと記述することができる。

例. 上記関数 fn の呼び出し（先の例の続き）

```
>>> fn(1,2) Enter ←通常の呼び出し形式
第 1 引数 x: 1
第 2 引数 y: 2
>>> fn(y=2,x=1) Enter ←仮引数の記号を指定した呼び出し
第 1 引数 x: 1 ←仮引数の記述の順序とは無関係に
第 2 引数 y: 2 正しく値を受け取っている
```

このような引数の形式は**キーワード引数**と呼ばれ、仮引数の記述とは異なる順序で実引数を与えることができる。

2.8.1.3 引数の個数が不定の関数

仮引数の個数が予め決まっていない関数も定義できる。それを実現する場合は、関数定義の仮引数にアスタリスク「*」で始まる名前を指定する。

書き方： `kansu(*args) { (定義内容) }`

このように記述すると、仮引数 `args` がタプルとして解釈される。すなわち関数内では、`args` は受け取った値を要素として持つタプルとなる。これを応用したプログラムの例を `test08-2.py` に示す。

プログラム：`test08-2.py`

```
1 # 個数未定の引数を取る関数
2 def argtest1( *a ):
3     print('引数に受け取ったもの:',a)
4     for m in a:
5         print( m )
6     return len(a)
7
8 # メインルーチン
9 n = argtest1('a',1,'b',2)
10 print('引数の個数',n)
```

これを実行すると次のような表示となる。

```
引数に受け取ったもの: ('a', 1, 'b', 2)
a
1
b
2
引数の個数 4
```

2.8.1.4 実引数に '*' を記述する方法

実引数に '*' を記述することができる。この場合は、'*' の直後に記述されたデータ構造の要素が、個々の引数として展開されて関数に渡される。これに関してはサンプルプログラム `argTest01.py` で動作を確認できる。

プログラム：`argTest01.py`

```
1 # テスト用関数
2 def f( *arg ):
3     print( '引数列:',arg )
4
5 # リストの全要素を引数列として渡す
6 a = ['x','y','z']; f( *a )
7
8 # タプルの全要素を引数列として渡す
9 a = ('s','t','u'); f( *a )
10
11 # 文字列を構成する文字を引数列として渡す
12 a = 'arguments'; f( *a )
```

このプログラムを実行すると次のような結果となる。

```
引数列: ('x', 'y', 'z')
引数列: ('s', 't', 'u')
引数列: ('a', 'r', 'g', 'u', 'm', 'e', 'n', 't', 's')
```

データ構造の各要素が個々の引数として関数に渡されていることがわかる。この記述方法は、多数の引数を関数に渡す際の簡便な方法となる

2.8.1.5 キーワード引数を辞書として受け取る方法

Python の関数ではキーワード引数が使え、次のプログラム例 `test08-3.py` について考える。

プログラム：`test08-3.py`

```
1 #--- キーワード引数を取る関数 ---
2 def argtest2( **ka ):
3     print( '名前: ', ka['name'] )
4     print( '年齢: ', ka['age'] )
```

```

5     print( '国籍: ', ka['country'] )
6     n = len(ka)
7     return n
8
9  #--- メインルーチン ---
10  a = argtest2( name='Tanaka', country='Japan', age=41 )
11  print('引数の個数: ',a,'\n')
12
13  # キーワード引数を辞書の形で与える方法
14  d = {'name':'John', 'country':'USA', 'age':'35'}
15  a = argtest2( **d )

```

このようにアスタリスク2つ「**」で始まる仮引数を記述すると関数側でその仮引数は辞書型オブジェクトとして扱える。(関数側ではキーは文字列型)

このプログラムの10行目に

```
argtest2( name='tanaka', country='japan', age=41 )
```

という形の関数呼び出しがある。キーワード引数を用いると「キーワード=値」という形式で引数を関数に渡すことができることを先に解説した。このプログラムを実行すると次のような表示となる。

```

名前: Tanaka
年齢: 41
国籍: Japan
引数の個数: 3

名前: John
年齢: 35
国籍: USA

```

2.8.1.6 実引数に「**」を記述する方法

先のプログラム test08-3.py の15行目に

```
a = argtest2( **d )
```

という形の関数呼び出しがある。これは一連のキーワード引数を辞書オブジェクトとして関数に与えるための方法であり、辞書オブジェクトの直前に「**」を記述し、それを実引数とする。

2.8.1.7 引数に与えたオブジェクトに対する変更の影響

関数が引数に受け取ったオブジェクトを変更する場合、その変更が元のオブジェクトに及ぼす影響について例を挙げて示す。次の例のように定義された関数 f は、引数として受け取ったものを関数内で変更する。

例. 引数に受け取ったものを関数内で変更する関数 f

```

>>> def f(nm,flt,c,strng,tpl,lst,dct,s): Enter
...     nm *= 2 Enter
...     print(' 関数内の nm:',nm) Enter
...     flt = flt**0.5 Enter
...     print(' 関数内の flt:',flt) Enter
...     c = 2+3j Enter
...     print(' 関数内の c:',c) Enter
...     strng = '新しい文字列' Enter
...     print(' 関数内の strng:',strng) Enter
...     tpl = (7,8,9) Enter
...     print(' 関数内の tpl:',tpl) Enter
...     lst[0],lst[-1] = lst[-1],lst[0] Enter
...     print(' 関数内の lst:',lst) Enter
...     dct['newKey'] = 'newValue' Enter
...     print(' 関数内の dct:',dct) Enter
...     s.add('z') Enter
...     print(' 関数内の s:',s) Enter
... Enter      ←関数定義の記述の終了
>>>           ←対話モードのプロンプトに戻った

```

次に関数 f の実行結果を観察する。まず各種のデータ型のオブジェクトを次のようにして用意する。

例. 各種オブジェクトの用意（先の例の続き）

```
>>> nm = 3      Enter      ←整数
>>> flt = 5.0    Enter      ←浮動小数点数
>>> c = 1+2j     Enter      ←複素数
>>> strng = '元の文字列'   Enter      ←文字列
>>> tpl = (4,5,6) Enter      ←タプル
>>> lst = ['a','b','c']    Enter      ←リスト
>>> dct = {'apple':'りんご','orange':'みかん'} Enter      ←辞書
>>> s = {'x','y'}          Enter      ←セット
```

この状況で関数 f を実行する。

例. 関数 f の実行（先の例の続き）

```
>>> f(nm,flt,c,strng,tpl,lst,dct,s) Enter      ←関数の呼び出し（実行）
関数内の nm: 6      ←関数内で変更された通りの値が出力されている
関数内の flt: 2.23606797749979      ↓
関数内の c: (2+3j)      :
関数内の strng: 新しい文字列
関数内の tpl: (7, 8, 9)
関数内の lst: ['c', 'b', 'a']
関数内の dct: {'apple': 'りんご', 'orange': 'みかん', 'newKey': 'newValue'}
関数内の s: {'y', 'x', 'z'}
```

関数 f が受け取った引数の内容を変更したものを出力していることがわかる。この処理の後で、引数に与えた変数の値を確認する例を次に示す。

例. 変数の値の確認（先の例の続き）

```
>>> nm      Enter
3      ←整数：変化していない
>>> flt      Enter
5.0      ←浮動小数点数：変化していない
>>> c      Enter
(1+2j)      ←複素数：変化していない
>>> strng      Enter
'元の文字列'      ←文字列：変化していない
>>> tpl      Enter
(4, 5, 6)      ←タプル：変化していない
>>> lst      Enter
['c', 'b', 'a']      ←リスト：変化している
>>> dct      Enter
{'apple':'りんご', 'orange':'みかん', 'newKey':'newValue'}      ←辞書：変化している
>>> s      Enter
{'y', 'x', 'z'}      ←セット：変化している
```

この例から、関数の引数に与えたオブジェクトの内容（値）が関数内での処理によってどのように影響を受けるかがわかる。上の例の結果から表 24 のようなことがわかる。

表 24: 関数内の処理が引数のオブジェクトに与える影響

引数に与えるオブジェクトの型	関数内部での変更が及ぼす影響
整数, 浮動小数点数, 複素数	関数内部でのみ引数の内容が変更される。
文字列, タプル	関数内部でのみ引数の内容が変更される。
リスト, 辞書, セット	関数内での変更処理が関数の実行終了後にも影響する。

2.8.1.8 引数に関するその他の事柄

■ 厳格な位置引数

関数の引数を厳格な形の位置引数として定義するには、関数定義の仮引数の記述の後にスラッシュ '/' を置く。

例. 厳格な位置引数を持つ関数 fn

```
>>> def fn(a,b,/):  ←仮引数の終わりに '/' を記述する
...     print('a =',a) 
...     print('b =',b) 
... 
>>> ← Python のプロンプトに戻った
```

この関数 fn を呼び出す例を次に示す。

例. 関数 fn を呼び出す試み（先の例の続き）

```
>>> fn(1,2)  ←正しい呼び出し
a = 1
b = 2

>>> fn(a=1,b=2)  ←キーワード引数を与えようとすると…
Traceback (most recent call last): ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: fn() got some positional-only arguments passed as keyword arguments: 'a, b'
```

この例からわかるように、キーワード引数を与えるとエラーが発生する。

■ 厳格なキーワード引数

関数の引数を厳格な形のキーワード引数として定義するには、関数定義の仮引数の記述の前にアスタリスク '*' を置く。

例. 厳格なキーワード引数を持つ関数 fk

```
>>> def fk(*,a):  ←仮引数の前に '*' を記述する
...     print(a) 
... 
>>> ← Python のプロンプトに戻った
```

この関数 fk を呼び出す例を次に示す。

例. 関数 fk を呼び出す試み（先の例の続き）

```
>>> fk(a=2)  ←正しい呼び出し
2

>>> fk(2)  ←位置引数を与えようとすると…
Traceback (most recent call last): ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: fk() takes 0 positional arguments but 1 was given
```

この例からわかるように、位置引数を与えるとエラーが発生する。また、期待されないキーワード引数を与えてもエラーとなる。

仮引数の '*' の替わりには不定個数の仮引数（p.139「2.8.1.3 引数の個数が不定の関数」で解説）を置いた場合も、その後に記述した仮引数は厳格なキーワード引数となる。

例. 不定個数の仮引数と厳格なキーワード引数を持つ関数 fk2

```
>>> def fk2(*p,a): 
...     print(p) 
...     print(a) 
... 
>>> ← Python のプロンプトに戻った
```

この関数 `fk2` を呼び出す例を次に示す。

例. 関数 `fk2` を呼び出す試み（先の例の続き）

```
>>> fk2(1,2,3,a=4)  Enter  ← fk2 の呼び出し
(1, 2, 3)
4
```

例. 関数 `fk2` の最後の引数の誤った与え方（先の例の続き）

```
>>> fk2(1,2,3,4)  Enter  ← fk2 の呼び出し（最後の引数が誤り）
Traceback (most recent call last):  ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: fk2() missing 1 required keyword-only argument: 'a'
```

■ 応用

上に説明した `('/', '*', 'a')` を組み合わせることができる。その場合は

```
def 関数名 ( 厳格な位置引数の並び, /, 通常の引数の並び, *, 厳格なキーワード引数の並び):
    (関数内部の記述)
```

と記述する。

■ 引数の暗黙値に関する事柄

関数の仮引数に設定された暗黙値は、その関数が呼び出される際に適用される。

例. 仮引数に暗黙値を持つ関数

```
>>> def f1( x=1 ):  Enter  ←関数定義の記述の開始
...     print(x)  Enter
...     x += 1  Enter  ←仮引数の値を更新する処理
...  Enter  ←関数定義の記述の終了
```

この後、上の関数 `f1` を複数回呼び出す様子を示す。（次の例）

例. 関数 `f1` の呼び出し毎に引数の値を確認する（先の例の続き）

```
>>> f1()  Enter  ←関数 f1 の呼び出し（1 回目）
1  ←引数 x は暗黙値となっている
>>> f1()  Enter  ←関数 f1 の呼び出し（2 回目）
1  ←この回でも引数 x は暗黙値となっている
```

仮引数に設定された暗黙値は、その関数の属性 `__defaults__` に、タプルの形で保持されている。（次の例）

例. 仮引数の暗黙値の確認（先の例の続き）

```
>>> f1.__defaults__  Enter  ←関数 f1 の仮引数の暗黙値の確認
(1,)  ←タプルの形で保持されている
```

暗黙値を持つ仮引数が複数ある場合は、`__defaults__` の要素に順番通り保持される。

▲注意▲

暗黙値がミュータブルなオブジェクトである場合、関数定義の内部でその内容を変更すると、それが `__defaults__` に反映される。（次の例）

例. 仮引数の暗黙値を変更する関数

```
>>> def f2( x=[] ):  Enter  ←仮引数の暗黙値がリストになっている
...     print(x)  Enter
...     x.append('a')  Enter  ←仮引数の暗黙値を変更している
...  Enter  ←関数定義の記述の終了
>>> f2.__defaults__  Enter  ←暗黙値の確認
([],)  ←仮引数に与えた暗黙値（空リスト）が保持されている
```

この定義の直後、仮引数 `x` の暗黙値の `id` 値を確認しておく。

例. 仮引数 x の暗黙値の id 値 (先の例の続き)

```
>>> id( f2.__defaults__[0] ) Enter ←仮引数 x の暗黙値の id 値を調べる
2660832482560 ← id 値 (これは実行時に決定される)
```

この値は後で使用する.

次に, 関数 f2 を複数回実行して, 仮引数 x の暗黙値を確認する. (次の例)

例. 仮引数の暗黙値の変化 (先の例の続き)

```
>>> f2() Enter ←関数 f2 の呼び出し (1 回目)
[] ←引数 x の当初の暗黙値
>>> f2() Enter ←関数 f2 の呼び出し (2 回目)
['a'] ←引数 x の暗黙値の内容が変化している
>>> f2() Enter ←関数 f2 の呼び出し (3 回目)
['a', 'a'] ←引数 x の暗黙値の内容が更に変化している
>>> f2.__defaults__ Enter ←この時点での暗黙値の確認
(['a', 'a', 'a'],) ←仮引数 x の暗黙値の最終的な値
```

関数 f2 の実行の度に仮引数の暗黙値が変化してしまっているように見えるが, その暗黙値の id 値を確認すると処理の前後で変化していない (同じオブジェクトである) ことがわかる. (次の例)

例. 仮引数 x の暗黙値の id 値を再度確認する (先の例の続き)

```
>>> id( f2.__defaults__[0] ) Enter
2660832482560 ← id 値 (関数 f2 定義直後と同じ)
```

結論. 関数の `__defaults__` 属性は, その関数の実行時に同一のオブジェクトを引数の暗黙値として与えるためのものである.

【仮引数にミュータブルオブジェクトを暗黙値として設定する安全な方法】

仮引数にミュータブルオブジェクトを安全な形で暗黙値として設定するには次のような工夫をすると良い.

例. 仮引数にミュータブルオブジェクトの暗黙値を与える安全な方法

```
>>> def f3( x=None ): Enter ←暗黙値として None を設定
...     if x is None: Enter ←実行時に引数が省略された場合は
...         x = [] Enter ←空リストを与える
...     print(x)
...     x.append('a') Enter ←引数 x を変更する処理
... Enter ←関数定義の記述の終了
```

少し余計な手間ではあるが, この例のように, 実行時に引数が省略されたことを None オブジェクトとして検知して, 当該引数にミュータブルオブジェクトを与えると安全である. (次の例)

例. 変化しない暗黙値の確認 (先の例の続き)

```
>>> f3() Enter ←関数 f3 の呼び出し (1 回目)
[] ←関数内の処理で設定された暗黙値
>>> f3() Enter ←関数 f3 の呼び出し (2 回目)
[] ←関数内の処理で設定された暗黙値 (先と変わらず)
>>> f3.__defaults__ Enter ←最終的な暗黙値
(None,) ←変化なし
```

p.142 で解説した「厳格なキーワード引数」にも暗黙値を設定することができる. その場合は, 当該関数オブジェクトの `__kwdefaults__` 属性にそれら暗黙値が保持される.

例. 厳格なキーワード引数に与える暗黙値

```
>>> def f(*,a,b,c=1,d=2): pass Enter ←厳格なキーワード引数に暗黙値を設定
... Enter ←関数定義の記述の終了
>>> f.__kwdefaults__ Enter ←厳格なキーワード引数の暗黙値の確認
{'c': 1, 'd': 2} ←辞書オブジェクトとして暗黙値が保持されている
```

厳格なキーワード引数に暗黙値を与える際は、それら引数をまとめて配置する必要はない。(次の例)

例. 暗黙値設定の柔軟な配置

```
>>> def f(*,a,b=1,c,d=2): pass Enter ←暗黙値を持つキーワード引数の順序は任意
... Enter ←関数定義の記述の終了
>>> f.__kwdefaults__ Enter ←暗黙値の確認
{'b': 1, 'd': 2} ←暗黙値の辞書ができています
```

▲注意▲

厳格なキーワード引数の暗黙値に関しても、位置引数の暗黙値と同様に、実行時の「同一オブジェクト」の再設定が行われる。すなわち、暗黙値としてミュータブルなオブジェクトを与えると、そのオブジェクトの内容は変更することができるので注意すること。

例. 暗黙値としてリスト（ミュータブルなオブジェクト）を与える例

```
>>> def f(*,a=[]): Enter ←厳格なキーワード引数にリストを設定
... print(a) Enter
... a.append('x') Enter ←キーワード引数暗黙値を変更
... Enter ←関数定義の記述の終了
>>> f.__kwdefaults__['a'] Enter ←キーワード引数 a の内容確認
[] ←暗黙値が設定されている
>>> id( f.__kwdefaults__['a'] ) Enter ←その id 値を調べる
2393438786816 ←id 値（実行時に決定される）
```

次に、関数 f を複数回実行して、仮引数 a の内容が変化の様子を確認する。(次の例)

例. 関数 f の実行による仮引数の変化（先の例の続き）

```
>>> f() Enter ←関数 f の呼び出し（1 回目）
[] ←引数 a の当初の暗黙値
>>> f() Enter ←関数 f の呼び出し（2 回目）
['x'] ←引数 a の暗黙値の内容が変化している
>>> f() Enter ←関数 f の呼び出し（3 回目）
['x', 'x'] ←引数 a の暗黙値の内容が更に変化している
```

この後、__kwdefaults__ 属性に設定された暗黙値を調べる。(次の例)

例. 仮引数 a の暗黙値とその id 値を再度確認する（先の例の続き）

```
>>> f.__kwdefaults__['a'] Enter ←仮引数 a の暗黙値の確認
['x', 'x', 'x'] ←最終的な値
>>> id( f.__kwdefaults__['a'] ) Enter ←その id 値の確認
2393438786816 ←当初のものと同じ（同一オブジェクトである）
```

以上のことから、厳格なキーワード引数の場合も、先の「仮引数にミュータブルオブジェクトを暗黙値として設定する安全な方法」(p.144) で解説した内容と同様の方法を取るべきである。

■ 引数部分に内包表記を与えた場合の動作

1 つの変数を取る関数の実行時に、実引数の部分に内包表記を与えると、それは 1 つのジェネレータ式¹⁴² を与えたことになる。

例. 引数での内包表記の記述

```
>>> def f(x): Enter
... print(x,':',type(x)) Enter ←引数の情報を出力し
... r = list(x) Enter ←引数からリストを作成し
... r.append('end') Enter ←末尾に 'end' を追加して
... return r Enter ←それを返す
... Enter ←関数定義の記述の終了
```

¹⁴² 「4.7.2 ジェネレータ式」(p.298) で解説する。

このように定義した関数 f を次のように実行すると、引数として何が渡されたかがわかる。

例. 関数 f の引数に内包表記を与えて実行する（先の例の続き）

```
>>> f( e for e in range(5) ) Enter ←引数に内包表記を与えると ↓受け取ったものはジェネレータ
<generator object <genexpr> at 0x0000021200281CC0> : <class 'generator'>
[0, 1, 2, 3, 4, 'end'] Enter ←戻り値
```

2.8.2 変数のスコープ（関数定義の内外での変数の扱いの違い）

関数の内部で生成したオブジェクトは基本的にはその関数の**ローカル変数**（局所変数）であり、その関数の実行が終了した後は消滅する。関数の外部で使用されている**グローバル変数**（大域変数）を関数内部で更新するには、それらを当該関数内でグローバル変数として宣言する必要がある。具体的には関数定義の内部で、

global **グローバル変数の名前**

と記述する。次に示すプログラム test08-4.py は、大域変数 gv の値が関数呼出しの前後でどのように変化するかを示す例である。

プログラム：test08-4.py

```
1  # 変数のスコープのテスト
2  gv = '初期値です, '      # 大域変数（グローバル変数）
3
4  #--- 関数内部で大域変数を使用する例 ---
5  def scopetest1():
6      global gv            # 大域変数であることの宣言
7      print('scopetest1 の内部では:',gv)
8      gv = 'scopetest1が書き換えたものです. '
9
10 #--- 大域変数と同名の局所変数を使用する例 ---
11 def scopetest2():
12     gv = 'scopetest2の局所変数gvの値です. '
13     print(gv)
14
15 #--- メインルーチン ---
16 print('【大域変数gvの値】 ')
17 print('scopetest1 呼び出し前:',gv)
18 scopetest1()
19 print('scopetest1 呼び出し後:',gv)
20 scopetest2()
21 print('scopetest2 呼び出し後:',gv)
```

このプログラムを実行した例を次に示す。

【大域変数 gv の値】

```
scopetest1 呼び出し前:  初期値です,
scopetest1 の内部では:  初期値です,
scopetest1 呼び出し後:  scopetest1 が書き換えたものです.
scopetest2 の局所変数 gv の値です.
scopetest2 呼び出し後:  scopetest1 が書き換えたものです.
```

注意)

Python とそれ以外の言語を比較すると、大域変数とローカル変数の扱いに違いが見られるので特に注意すること。先に global の宣言により関数内部で大域変数を使用することを述べたが、この宣言をしなくても**関数内部で大域変数の参照のみは可能**であることも注意すべき点である。安全なコーディングのために推奨される指針を以下にいくつか挙げる。

- ・ 関数に値を渡すには引数を介して行い、関数内部から外部への値は return を用いて返すことを基本とする。
- ・ 関数内部で使用する変数の大域／ローカルの区別を強く意識し、可能な限り、関数の内部と外部で異なる変数名を使用する。
- ・ 関数内部で大域変数を使用する場合は、その関数の定義の冒頭で global 宣言する。（あまり推奨されない）
- ・ 関数内部でのみ使用するローカル変数については、その関数の定義の冒頭で値の設定をしておく。特に設定すべき値がなくとも何らかの初期値を強制的に与えておくのが良い。

2.8.3 関数の再帰的定義

Python では関数を再帰的に定義することができる。例えば、0 以上の整数 n の階乗 $n!$ は次のように再帰的に定義できる。

$$n! = \begin{cases} n = 0 & \rightarrow 1 \\ n > 0 & \rightarrow n \times (n-1)! \end{cases}$$

これを Python の関数 `fct` として次のように定義することができる。

例. $n!$ を関数 `fct` として実装する

```
>>> def fct(n):  ← 定義の記述の開始
...     if n == 0: 
...         return 1 
...     else: 
...         return n*fct(n-1) 
...  ← 定義の記述の終了
>>>  ← 対話モードのプロンプトに戻った
```

この関数を評価する例を次に示す。

例. $10!$ の計算（先の例の続き）

```
>>> fct(10)  ←  $10!$  の計算
3628800  ← 計算結果
```

正しく計算できていることがわかる。

2.8.3.1 再帰的呼び出しの回数の上限

関数を再帰的に呼び出す深さ（再帰的呼び出し回数）には上限があり、それを超えて再帰的呼び出しを実行することはできない。（次の例参照）

例. $1000!$ の計算：深すぎる再帰的呼び出し（先の例の続き）

```
>>> fct(1000)  ←  $1000!$  の計算を試みると…
Traceback (most recent call last):  ← 再帰的呼び出しの上限を超えているので実行できない
  File "<stdin>", line 1, in <module>  ← ことを意味するエラーが発生する。
  File "<stdin>", line 5, in fct
  File "<stdin>", line 5, in fct
  File "<stdin>", line 5, in fct
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

再帰的呼び出し回数の上限は `sys` モジュールの `getrecursionlimit` で調べることができる。

例. 再帰的呼び出し回数の上限を調べる（先の例の続き）

```
>>> import sys  ← sys モジュールの読み込み
>>> sys.getrecursionlimit()  ← 再帰的呼び出し回数の上限を調べる
1000  ← 呼び出し回数は 1000 未満に制限されていることがわかる
```

先の例で試みた $1000!$ の計算では関数 `fct` の呼び出し回数が 1000 を超えているためエラーとなった。

`sys` モジュールの `setrecursionlimit` を用いると、関数の再帰的呼び出し回数の上限を変更することができる。

例. 再帰的呼び出し回数の上限を大きくする（先の例の続き）

```
>>> sys.setrecursionlimit(2048)  ← 再帰的呼び出し回数を大きく設定 (2,048) する
```

この後で関数 `fct` を評価すると次のように正しく実行される。

例. 関数 fct を再度実行 (先の例の続き)

```
>>> fct(1000)  [Enter]    ← 1000! の計算を試みると…
40238726007709377354370243392300398571937486421071463254379991042993851
23986290205920442084869694048004799886101971960586316668729948085589013
23829669944590997424504087073759918823627727188732519779505950995276120
⋮
(途中省略)
⋮
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
```

1000! の計算結果が得られていることがわかる。

補足) 再帰的関数呼び出しの理解を深めるための考察

先に示した関数 fct がその内部で次々と自分自身を呼び出して最終的に $n == 0$ となるまでの様子を次のようにして眺める。

例. 関数の処理の開始時と終了時にメッセージを出力する関数 fct2

```
>>> def fct2(n):  [Enter]
...     print('entered with',n)  [Enter]
...     if n==0: r=1  [Enter]
...     else: r=n*fct2(n-1)  [Enter]
...     print('exited with',r)  [Enter]
...     return r  [Enter]
...  [Enter]
>>>  ← Python のプロンプトに戻った
```

このように定義された関数 fct2 は先の fct と本質的には変わらないが、関数が呼び出された時点と関数の処理が終了する時点でメッセージを出力する形となっている。この fct2 を実行する例を次に示す。

例. 関数 fct2 の実行 (先の例の続き)

```
>>> fct2(3)  [Enter]
entered with 3    ← fct2(3) の処理開始
entered with 2    ← fct2(2) の処理開始
entered with 1    ← fct2(1) の処理開始
entered with 0    ← fct2(0) の処理開始
exited with 1     ← fct2(0) の処理終了
exited with 1     ← fct2(1) の処理終了
exited with 2     ← fct2(2) の処理終了
exited with 6     ← fct2(3) の処理終了
6    ←最終的な計算結果
```

関数 fct2 がその内部で次々と関数 fct2 を呼び出している様子がわかる。

2.8.3.2 再帰的呼び出しを応用する際の注意

関数の再帰的呼び出しを応用する際には、先に示したような呼び出し回数の上限の問題や、計算量や記憶資源に関する問題についても注意する必要がある。これに関しても例を挙げて説明する。

0 以上の整数 n に対するフィボナッチ数 F_n は次のように再帰的に定義することができる。

$$F_n = \begin{cases} n = 0 & \rightarrow 0 \\ n = 1 & \rightarrow 1 \\ n > 1 & \rightarrow F_{n-1} + F_{n-2} \end{cases}$$

これを再帰的な関数呼び出しで実装する次のようなプログラム fibonacci01.py について考える。

プログラム: fibonacci01.py

```
1 import time
2 # フィボナッチ数を算出する関数
3 def fib(n):
4     if n == 0:
5         return 0
```

```

6         elif n == 1:
7             return 1
8         else:
9             return fib(n-1)+fib(n-2)
10
11 # F0~F19 までのフィボナッチ数列
12 print('F0-F19',[fib(n) for n in range(20)])
13
14 # F0~F35 までのフィボナッチ数列
15 t1 = time.time()
16 print('\nF0-F35',[fib(n) for n in range(36)])
17 t2 = time.time()
18 print('計算時間:',t2-t1,'(sec)')
```

このプログラムではフィボナッチ数を算出する関数 `fib` を再帰的関数呼び出しで実装している。12 行目ではこの関数を用いて $F_0 \cdots F_{19}$ の数列を作成して表示している。この部分の実行には大きな時間はかからないが、16 行目の $F_0 \cdots F_{35}$ の数列の作成と表示にはかなり時間がかかる。この部分の実行時間を `time.time()` を用いて¹⁴³ 計測して表示する。

このプログラムを実行した例を次に示す。

実行例. `fibonacci01.py` を実行した例 (Intel Corei7-6670HQ, 2.6GHz, RAM 16GB, Windows10 Pro の環境で実行)

```

F0-F19 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
F0-F35 [0, 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]
計算時間: 9.258679389953613 (sec)
```

上のプログラム `fibonacci01.py` では関数 `fib(n)` の計算量は 2^n に比例する形となり¹⁴⁴ 計算時間が大きくなる。従って、この形での実装は好ましくないことがわかる。

課題. 関数の再帰的呼び出しに依らない形でフィボナッチ数を算出する関数 `fib2` を実装せよ。

ヒント. リストなどのデータ構造に $F_0 \cdots F_n$ を順次生成してゆく方法などがある。

2.8.4 内部関数

関数定義の記述の中に、別の関数定義を記述することができる。これに関してサンプルプログラム `innerFunc01.py` を挙げて説明する。

プログラム: `innerFunc01.py`

```

1 # coding: utf-8
2 #--- 関数定義 ---
3 def f1( x ):
4     y = 2*x
5     #--- 内部関数 (ここから) ---
6     def f2( a ):
7         y = 3*a
8         print( 'f2内部のy:', y )
9         return y
10    #--- 内部関数 (ここまで) ---
11    f2( y )
12    print( 'f1内部のy:', y )
13    return y
14
15 #--- メイン ---
16 print('● f1(5)の実行')
17 f1(5)
18 # エラーとなる実行
19 print('● f2(5)の実行を試みると...')
20 f2(5)
```

¹⁴³後の「4.1.2.4 時間の計測」(p.240)で解説する。

¹⁴⁴ $O(2^n)$ の規模

このプログラムでは、関数 `f1` の定義の記述の中に、別の関数 `f2` が定義されている。この場合、関数 `f2` は関数 `f1` の内部でのみ呼び出す（実行する）ことができる。この例における関数 `f2` は、関数 `f1` の**内部関数**である。

このプログラムを実行した例を次に示す。

```
● f1(5) の実行
f2 内部の y: 30
f1 内部の y: 10
● f2(5) の実行を試みると…
Traceback (most recent call last):
  File "C:\Users\katsu\innerFunc01.py", line 21, in <module>
    f2(5)
    ^^
NameError: name 'f2' is not defined. Did you mean: 'f1'?
```

この例からもわかるように、プログラムのメイン部分から直接関数 `f2` を呼び出すことができません、それを試みると上記のようなエラーが発生する。

2.8.4.1 内部関数におけるローカル変数

先の例 `innerFunc01.py` の関数 `f2` の内部では、`f2` のためのローカル変数 `y` が使用されている。この `y` のスコープは `f2` 内部に限られ、関数 `f1` 直下のローカル変数 `y` とは別のものである。このことが先の実行例から理解できる。

内部関数の内側でその上の関数の変数を使用するには、内部関数側で `nonlocal` 宣言する。これに関してサンプルプログラム `innerFunc02.py` を例に挙げて考える。

プログラム：innerFunc02.py

```
1  # coding: utf-8
2  #--- 関数定義 ---
3  def f1( x ):
4      y = 2*x
5      #--- 内部関数（ここから）---
6      def f2( a ):
7          nonlocal y  # 上の階層の変数 y の使用
8          y += a
9          print( 'f2内部のy:', y )
10     #--- 内部関数（ここまで）---
11     f2( 3 )
12     print( 'f1内部のy:', y )
13     return y
14
15 #--- メイン ---
16 print( '● f1(5) の実行' )
17 f1(5)
```

このプログラムの7行目で変数 `y` を `nonlocal` 宣言している。これにより、関数 `f2` 内部の変数 `y` は上位の関数 `f1` のものと同じものになる。

このプログラムを実行した例を次に示す。

```
● f1(5) の実行
f2 内部の y: 13
f1 内部の y: 13
```

関数 `f1`, `f2` の両方において変数 `y` は同じものであることがわかる。

2.8.4.2 内部関数はいつ作成されるのか

内部関数は、それを含む上位の関数が評価（実行）された時点で生成される。このことについて次の例で確かめる。

例. 内部関数 f2 を持つ関数 f1

```
>>> def f1(): Enter ←関数 f1 の定義の開始
...     def f2(): Enter ←内部関数 f2 の定義の開始
...     pass Enter ←関数 f2 の定義の記述を省略
...     return f2 Enter ←関数 f1 は内部関数 f2 の関数オブジェクトを返す
... Enter ←関数 f1 の定義の終了
>>> ← Python のプロンプトに戻った
```

この関数 f1 は、その内部関数 f2 の関数オブジェクトを返すものである。(次の例)

例. 関数 f1 の評価 (先の例の続き)

```
>>> f1() Enter ←関数 f1 の評価 (実行)
<function f1.<locals>.f2 at 0x000001D373DC2B88> ←内部関数 f2 の関数オブジェクト
```

次に、f1 の評価を複数回実行した戻り値をリストにして、その要素を確認する。

例. 複数の関数 f1 の戻り値を確認 (先の例の続き)

```
>>> for r in [f1(),f1(),f1()]: Enter ←関数 f1 の評価を 3 回実行した結果を確認
...     print( r ) Enter ←各値を出力
... Enter ←繰り返しの記述の終了
<function f1.<locals>.f2 at 0x000001D373DC2828> ← 1 回目の評価結果
<function f1.<locals>.f2 at 0x000001D373DF60D8> ← 2 回目の評価結果
<function f1.<locals>.f2 at 0x000001D373DF6168> ← 3 回目の評価結果
```

この実行例から、内部関数 f2 は上位の関数 f1 を評価する毎に異なるものとして定義されていることがわかる。

2.8.4.3 内部関数の応用：クロージャ、エンクロージャ

関数はその処理が終了すると、実行の制御を呼び出し元に返し、関数内部のローカル変数などは廃棄されるので、関数の実行の前後で関数内の状態を保存することができない。関数内の状態を保存するにはクロージャとエンクロージャという機構を構成する方法がある。これは内部関数と変数の nonlocal 宣言を用いて実現できる。これに関して例を示して説明する。

次のように、内部関数 clsr を持つ関数 encl を考える。

例. 内部関数 clsr を持つ関数 encl の定義

```
>>> def encl(): Enter ←外側の関数 encl の記述の開始
...     x = [ ] Enter ←関数 encl のローカル変数
...     def clsr( n ): Enter ←内部関数 clsr の記述の開始
...         nonlocal x Enter ←外側の関数 encl の変数を使用する宣言
...         x.append(n) Enter ←その末尾に clsr が受け取った値を追加する処理
...         return x Enter ←処理結果を返す
...     return clsr Enter ←上のように定義された内部関数を返す
... Enter ←外側の関数 encl の記述の終了
>>> ← Python のプロンプトに戻った
```

このように定義された関数 encl は内部関数 clsr を生成して返す。(次の例)

例. 関数 encl を実行して内部関数を取得する (先の例の続き)

```
>>> c1 = encl() Enter ←内部関数を c1 として受け取る
```

この処理により、c1 が関数として実行できる。(次の例)

例. 関数 c1 の実行 (先の例の続き)

```
>>> c1(1) Enter ←関数 c1 の実行
[1] ←戻り値
```

関数 c1 の実行結果として [1] が得られているが、これは外側の関数 encl のローカル変数 x に保存されているものである。ここで注目すべき点が、「関数 encl は既に終了しているにも関わらず、そのローカル変数 x の内容が保持されている」ということである。通常の場合は関数の実行が終了すると、そのローカル変数は廃棄されるが、上の例で

は、関数 `enc1` のローカル変数 `x` を関数 `c1` がまだ参照している（参照カウントが0でない）ので `enc1` 終了後も存在している。（次の例）

例. 引き続き使用可能な変数 `x`（先の例の続き）

```
>>> c1(2) Enter ←引き続き c1 を実行
[1, 2]          ←変数 x の値が得られている
>>> c1(3) Enter ←更に c1 を実行
[1, 2, 3]       ←変数 x の値が得られている
```

当然であるが、再度 `enc1` を実行して内部関数を別の関数オブジェクトとして生成すると得られた内部関数は先に作成した `c1` とは別の状態を保持する。（次の例）

例. 再度関数 `enc1` を実行して内部関数を生成する（先の例の続き）

```
>>> c2 = enc1( ) Enter ←新たな内部関数 c2 を作成
>>> c2(100) Enter      ←それを実行
[100]                  ←戻り値 (c2 が独自に持つ変数 x の値)
>>> c2(101) Enter      ←続けて c2 を実行
[100, 101]             ←戻り値 (c2 が独自に持つ変数 x の値)
>>> c1(4) Enter         ←先の c1 を実行すると…
[1, 2, 3, 4]           ← c1 が独自に持つ変数 x の値が得られている
```

今回の例で示した「外側の関数」`enc1` は内部関数 `clsr` の内部状態を保持する**エンクロージャ**として働く。またこの場合の内部関数 `clsr` はエンクロージャの状態を使用する**クロージャ**として働く。

エンクロージャは複数のクロージャを生成する形で実装することもできる。次に示す例は、2つのクロージャを持つエンクロージャを実装するものである。

例. 2つのクロージャを持つエンクロージャ

```
>>> def enc(): Enter ←エンクロージャ
...     x = 0 Enter ←クロージャが共有する変数
...     def inc(): Enter ←クロージャ(1)
...         nonlocal x Enter
...         x += 1 Enter
...         return x Enter
...     def dec(): Enter ←クロージャ(2)
...         nonlocal x Enter
...         x -= 1 Enter
...         return x Enter
...     return (inc,dec) Enter クロージャを返す
... Enter ←エンクロージャの記述の終了
>>> Enter ←Python のプロンプトに戻った
```

この関数 `enc` は2つのクロージャを返すエンクロージャで、それらクロージャが変数 `x` を共有する。クロージャ `inc` は変数 `x` の値を1増やし、`dec` は変数 `x` の値を1減らすものである。

例. 上記関数を使用する（先の例の続き）

```
>>> f1, f2 = enc() Enter ←クロージャを作成
>>> [f1(),f1(),f1()] Enter ←クロージャ(1) を3回実行
[1, 2, 3]                  ←実行結果 (1)
>>> [f2(),f2(),f2()] Enter ←クロージャ(2) を3回実行
[2, 1, 0]                  ←実行結果 (2)
```

2つのクロージャがエンクロージャのローカル変数 `x` を共有している様子がわかる。

内部状態を保持する関数の実装と同様のことを実現する（更に高機能な）方法として、後に説明する**オブジェクト指向プログラミング**がある。これに関しては「2.9 オブジェクト指向プログラミング」(p.154) で解説する。

2.8.5 定義した関数の削除

def 文で定義した関数は del 文によって削除することができる。(次の例参照)

例. 関数の定義と削除

```
>>> def dbl(x): return 2*x  Enter    ←与えた数の2倍を算出する関数 dbl の定義
...  Enter    ←定義の記述の終了
>>> dbl(3)  Enter    ←3の2倍を算出する
6          ←計算結果
>>> del dbl  Enter    ←関数 dbl の削除
>>> dbl(3)  Enter    ←先と同じ計算を試みると…
Traceback (most recent call last):      ←関数 dbl が定義されていないことによるエラー
  File "<stdin>", line 1, in <module>    ←メッセージが表示される.
NameError: name 'dbl' is not defined    ←(関数 dbl が削除されたことがわかる)
```

この例は、一旦定義された関数 dbl が del 文によって削除されることを示すものである。削除した後に当該関数を呼び出そうとすると、未定義のため実行できない旨のエラーが発生する。

2.9 オブジェクト指向プログラミング

Python におけるオブジェクト指向の考え方も他の言語のそれと概ね同じである。ここでは、オブジェクト指向についての基本的な考え方の説明は割愛して、**クラスやメソッドの定義方法やインスタスの取り扱い**といった具体的な事柄について説明する。

2.9.1 クラスの定義

《 class の記述 》 その 1

書き方: class クラス名:
(定義の記述)

「定義の記述」は class よりも右の位置に同一の深さのインデント（字下げ）を施してスイートとして記述する。

別のクラスをスーパークラス（上位クラス，**基底クラス**などと呼ぶこともある）とし，その**拡張クラス**（サブクラス，**派生クラス**と呼ぶこともある）としてクラスを定義するには次のように記述する．

《 class の記述 》 その 2

書き方: class クラス名 (スーパークラス):
(定義の記述)

「スーパークラス」はコンマで区切って複数記述することができる。

Python では**多重継承**が可能である。拡張クラスがスーパークラスの定義を引き継ぐことを**継承**という。

上記「その 1」のようにスーパークラスを指定せずにクラスを定義すると、そのクラスは object クラスの拡張クラスとなる。object クラスは Python 言語における最上位のクラスである。Python 言語のクラス階層については後の「4.30 Python の型システム」(p.362)で解説する。

2.9.1.1 コンストラクタとファイナライザ

クラスのインスタンスを生成する際のコンストラクタは、クラスの定義内に次のように `__init__` を記述する。(init の前後にアンダースコアを 2 つ記述する)

《 __init__ の記述 》

書き方: `def __init__(self, 仮引数):`
(定義の記述)

仮引数はコンストラクタを呼び出す際に与えられる実引数を受け取るものである。仮引数は複数記述することができる。また仮引数は省略可能である。「定義の記述」は `def` よりも右の位置に同一の深さのインデント（字下げ）を施してスイートとして記述する。`self` は生成するインスタンス自身を指しており、第 1 仮引数に記述する。ただし、コンストラクタを呼び出す際には引数に `self` は記述しない。

スーパークラスを持つクラスのコンストラクタ内では

`super().__init__(引数の列)`

を記述¹⁴⁵ して、スーパークラスのコンストラクタを呼び出すことができる。コンストラクタはクラスのインスタンスを生成する際に実行される初期化処理である。インスタンスの生成は

インスタンス名 = クラス名 (引数)

とする。「引数」にはコンストラクタの `__init__` に記述した `self` より右の仮引数に与えるものを記述する。

クラスのインスタンスは、使用されなくなると自動的に廃棄される¹⁴⁶。この際の処理をファイナライザ¹⁴⁷として記述することができる。ファイナライザは `__del__` の名前で定義する。

¹⁴⁵ 詳しくは後の「2.9.9 メソッドのオーバーライドと super()」(p.173)で解説する。

¹⁴⁶ガベージコレクション (GC : Garbage Collection)

¹⁴⁷ デストラクタと呼ばれることもある。

《 __del__ の記述 》

書き方： `def __del__(self):`
(定義の記述)

「定義の記述」は `def` よりも右の位置に同一の深さのインデント（字下げ）を施してスイートとして記述する。
`self` は廃棄するインスタンス自身を指しており、第 1 仮引数に記述する。

スーパークラスを持つクラスのファイナライザ内では

```
super().__del__( )
```

を記述して、スーパークラスのファイナライザを呼び出すことができるが、これは推奨されない。拡張クラスがスーパークラスのファイナライザを呼び出すと、ファイナライザの連鎖の動きがプログラマにとって予想しにくいものとなることがあるというのが理由である。

■ インスタンスの生成と廃棄に関するサンプルプログラム

クラスのインスタンスが生成あるいは廃棄される際の動作を示すサンプルプログラム `ConstFinal01.py` を次に示す。

プログラム： `ConstFinal01.py`

```
1 # coding: utf-8
2 #--- クラス定義 ---
3 class C:
4     def __init__(self):
5         print('クラスCのインスタンス',self,'が作成されました。')
6     def __del__(self):
7         print('クラスCのインスタンス',self,'が廃棄されました。')
8
9 #--- インスタンスの作成と廃棄 ---
10 print('クラスCのインスタンスを生成します。')
11 x = C()
12 print('クラスCのインスタンスを廃棄します。')
13 x = None
14
15 #--- 終了待ち ---
16 s = input('Enterで終了します>')
```

解説：

このプログラムは、クラス `C` のインスタンス `x` を生成し、それを廃棄する例を示すものである。11 行目でクラス `C` のインスタンスが生成されて変数 `x` に与えられているが、この処理の際にコンストラクタが働き、インスタンスが生成されたことを通知するメッセージを表示する。

13 行目では、変数 `x` に `None` が代入されており、先に保持されていたクラス `C` のインスタンスは自動的に廃棄される。この処理の際にファイナライザが働き、インスタンスが廃棄されたことを通知するメッセージを表示する。

このプログラムは最後に `Enter` を押すことで終了する。このプログラムを実行した例を次に示す。

```
クラスCのインスタンスを生成します。
クラスCのインスタンス <__main__.C object at 0x000001CDDEBCBD88> が作成されました。
クラスCのインスタンスを廃棄します。
クラスCのインスタンス <__main__.C object at 0x000001CDDEBCBD88> が廃棄されました。
Enter で終了します>
```

次に、スーパークラスを持つクラスのインスタンスを生成、廃棄するサンプルプログラム `ConstFinal02.py` を示す。

プログラム： `ConstFinal02.py`

```
1 # coding: utf-8
2 #--- クラス定義 ---
3 class B:                                # 基底クラスB
4     def __init__(self):
5         print('基底クラスBのコンストラクタが呼び出されました。')
6     def __del__(self):
7         print('基底クラスBのファイナライザが呼び出されました。')
8
9 class C(B):                             # 拡張クラスC
```

```

10     def __init__(self):
11         super().__init__()
12         print('拡張クラスCのインスタンス',self,'が作成されました。')
13     def __del__(self):
14         super().__del__()
15         print('拡張クラスCのインスタンス',self,'が廃棄されました。')
16
17 #--- インスタンスの作成と廃棄 ---
18 print('クラスCのインスタンスを生成します。')
19 x = C()
20 print('クラスCのインスタンスを廃棄します。')
21 x = None
22
23 #--- 終了待ち ---
24 s = input('Enterで終了します>')

```

解説：

このプログラムでは、クラス B とその拡張クラス C が定義されており、クラス C のインスタンスを生成、廃棄する際に、スーパークラス（基底クラス）である B のコンストラクタ、ファイナライザを呼び出している。

このプログラムを実行した例を次に示す。

```

クラスCのインスタンスを生成します。
基底クラスBのコンストラクタが呼び出されました。
拡張クラスCのインスタンス <__main__.C object at 0x0000024FA571AF88> が作成されました。
クラスCのインスタンスを廃棄します。
基底クラスBのファイナライザが呼び出されました。
拡張クラスCのインスタンス <__main__.C object at 0x0000024FA571AF88> が廃棄されました。
Enter で終了します>

```

このプログラムは最後に `Enter` を押すことで終了する。

2.9.1.2 インスタンス変数

クラスのインスタンスとしてのオブジェクトは**インスタンス変数**¹⁴⁸（属性：attribute、プロパティ:property と呼ぶこともある）を持ち、それらが各種の値を保持する。インスタンス変数へのアクセスは

(1) 参照時： インスタンス名. 変数名

(2) 設定時： インスタンス名. 変数名 = 値

とする。基本的には (2) の形で随時インスタンス変数を追加することができる。また Python では、クラスのコンストラクタの中で

```
self.変数名 = 値
```

と記述してインスタンス変数を初期化するのが一般的である。

2.9.1.3 メソッドの定義

クラスの定義内容としてメソッドの定義を記述する。これは関数の定義の記述とよく似ている。インスタンスメソッド（クラスのインスタンスに対するメソッド）の定義は次のように記述する。

《メソッドの定義1》 インスタンスに対するメソッド

書き方： `def メソッド名 (self, 仮引数):`
(定義の記述)

仮引数は複数記述することができる。また仮引数は省略可能である。「定義の記述」は `def` よりも右の位置に同一の深さのインデント（字下げ）を施してスイートとして記述する。self は対象となるインスタンス自身を指しており、第 1 仮引数に記述する。当該メソッド呼び出し時には引数には self は記述しない。

インスタンスメソッドとは別に**クラスメソッド**を定義することができ、それは

クラス名. クラスメソッド (引数並び)

と記述して実行する。クラスメソッドの定義は次のように記述する。

¹⁴⁸Python 以外の言語では「メンバ」と呼ばれることがある。

《メソッドの定義2》 クラスメソッド

書き方: `@classmethod`

```
def メソッド名 ( cls, 仮引数 ):
    (定義の記述)
```

仮引数は複数記述することができる。また仮引数は省略可能である。「定義の記述」は `def` よりも右の位置に同一の深さのインデント（字下げ）を施してスイートとして記述する。`cls` は当該クラスを指しており、第1仮引数に記述する。当該メソッド呼び出し時には引数には `cls` は記述しない。

クラスメソッド（クラス自体に対するメソッド）を定義するには、`def` の1行前に `def` と同じインデント位置に「`@classmethod`」というデコレータ¹⁴⁹を記述する。

継承関係のあるクラスの間で、拡張クラス側にスーパークラスのメソッドと同じ名前のメソッドを定義するとオーバーライドされ、拡張クラス側のメソッドが優先される。

2.9.1.4 クラス変数

インスタンスの変数ではなく、クラスそのものに属する変数をクラス変数という。クラス変数は、値の設定などをクラス定義の中で記述することで作成する。

一般的なオブジェクト指向の考え方では、クラス内にメンバと呼ばれるオブジェクトが定義され、そのクラスのインスタンスが個別に持つ属性の値を保持する。Pythonではメンバの存在を明に宣言するのではなく、そのクラスのコンストラクタやメソッドの定義の中で、`self` や `cls` の属性として値を与える（オブジェクトを生成する）ことでインスタンス変数やクラス変数を生成する。

【サンプルプログラム】

クラス、クラス変数、インスタンス、インスタンス変数、インスタンスに対するメソッド、クラスメソッドの使用方法を簡単に示すサンプルを `test09.py` に示す。

プログラム: `test09.py`

```
1  # coding: utf-8
2  # クラスに関する試み
3  class testClass:      #--- クラス定義 ---
4      classV = 'testClassのクラス変数'
5      # コンストラクタ
6      def __init__(self, v):
7          self.instanceV = 'インスタンスの変数:' + str(v)
8      # クラスメソッド
9      @classmethod
10     def cMethod( cls, a ): # 第1仮引数には当該クラスが与えられる
11         print( '引数 cls:', cls )
12         print( '引数 a:', a )
13         print( 'クラス変数 classV:', cls.classV )
14         print( '' )
15     # インスタンスに対するメソッド
16     def iMethod( self, a ): # 第1仮引数には当該インスタンスが与えられる
17         print( '引数 self:', self )
18         print( '引数 a:', a )
19         print( 'インスタンス変数 instanceV:', self.instanceV )
20         print( 'クラス変数 classV:', testClass.classV )
21         print( '' )
22 #--- インスタンスの生成 ---
23 m1 = testClass( 1 )
24 m2 = testClass( 2 )
25 #--- インスタンスに対するメソッドの実行 ---
26 m1.iMethod( 1 )
27 m2.iMethod( 2 )
28 #--- クラスメソッドの実行 ---
29 testClass.cMethod( 3 )
```

このプログラムを実行した例を次に示す。

¹⁴⁹Javaのアノテーションとよく似た働きをする記述であるが、詳しくは「4.22 デコレータ」(p.344)を参照のこと。

```

引数 self: <__main__.testClass object at 0x0000023D31437F28>
引数 a: 1
インスタンス変数 instanceV: インスタンスの変数:1
クラス変数 classV: testClass のクラス変数

引数 self: <__main__.testClass object at 0x0000023D31437C50>
引数 a: 2
インスタンス変数 instanceV: インスタンスの変数:2
クラス変数 classV: testClass のクラス変数

引数 cls: <class '__main__.testClass'>
引数 a: 3
クラス変数 classV: testClass のクラス変数

```

インスタンス変数が個別に値を保持している様子がわかる。またクラス変数はクラス自体に属する形で存在する様子がわかる。

もう1例、オブジェクト指向のプログラム test09-1.py を次に示す。

プログラム：test09-1.py

```

1  # coding: utf-8
2  # クラス定義
3  class Person:
4      # クラス変数
5      population = 7400000000
6      # コンストラクタ
7      def __init__(self, Name, Age, Gender, Country):
8          self.name = Name
9          self.age = Age
10         self.gender = Gender
11         self.country = Country
12     # クラスメソッド
13     @classmethod
14     def belongTo(cls):
15         print(cls, 'は人類に属しています。')
16         return 'Human'
17     # メソッド
18     def getName(self):
19         print('名前は', self.name, 'です。')
20         return self.name
21     def getAge(self):
22         print('年齢は', self.age, '才です。')
23         return self.age
24     def getGender(self):
25         print('性別は', self.gender, '性です。')
26         return self.gender
27     def getCountry(self):
28         print(self.country, 'から来ました。')
29         return self.country
30
31 # メインルーチン
32 m1 = Person('太郎', 39, '男', '日本')
33 m2 = Person('アリス', 28, '女', 'アメリカ')
34
35 m1.getName()
36 m1.getAge()
37 m1.getGender()
38 m1.getCountry()
39
40 m2.getName()
41 m2.getAge()
42 m2.getGender()
43 m2.getCountry()
44
45 Person.belongTo()
46 print('現在の人口は約', Person.population, '人です。')

```

このプログラムを実行すると、次のように表示される。

名前は 太郎 です.
 年齢は 39 才です.
 性別は 男 性です.
 日本 から来ました.
 名前は アリス です.
 年齢は 28 才です.
 性別は 女 性です.
 アメリカ から来ました.
 <class '__main__.Person'> は人類に属しています.
 現在の人口は約 7400000000 人です.

参考) Python では**スタティックメソッド** (静的メソッド, staticmethod) というものも扱える. これに関しては後の「2.9.11.6 静的メソッド (スタティックメソッド)」(p.181) で解説する.

2.9.2 クラス, インスタンス間での属性の関係

「**インスタンス.クラス変数**」の形式でクラス変数を参照することができる. またインスタンスに対してクラスメソッドを実行することができる. このことについて例を挙げて解説する. まずサンプルとして, 次のようなクラス C を定義する.

例. サンプルのクラス定義

```

>>> class C:
...     cv = 'クラス変数の値'
...     @classmethod
...     def cm(cls):
...         print('クラスメソッドからの出力')
...     def __init__(self):
...         self.iv = 'インスタンス変数の値'
...     def im(self):
...         print('インスタンスメソッドからの出力')
...
>>>
  
```

← Python のプロンプトに戻った

まず当然のことではあるが, インスタンス変数, クラス変数が参照できる. (次の例)

例. インスタンス変数, クラス変数の参照 (先の例の続き)

```

>>> i = C()
>>> i.iv
>>> C.cv
  
```

←インスタンス i の作成
 ←インスタンス変数の確認
 'インスタンス変数の値'
 ←クラス変数の確認
 'クラス変数の値'

次に, **インスタンス.クラス変数** の形式でクラス変数を参照できることを示す.

例. クラス変数の参照 (先の例の続き)

```

>>> i.cv
  
```

←クラス変数の参照
 'クラス変数の値' ←正しく参照できている

ただしこれは参照のみであり, i.cv に値を代入すると cv は i のインスタンス変数となる. (次の例)

例. 代入の例 (先の例の続き)

```

>>> i.cv = 'インスタンス i の変数 cv の値'
>>> i.cv
>>> C.cv
  
```

←これにより cv はインスタンス変数となる
 ←内容確認
 'インスタンス i の変数 cv の値'
 ←正しい形でクラス変数を確認すると
 'クラス変数の値' ←元のままである

クラス変数は変更されておらず, インスタンス変数への代入の影響を受けていないことがわかる.

次にクラスメソッド、インスタンスメソッドについて考察する。当然のことではあるが、クラス名に対してクラスメソッドを、インスタンスに対してインスタンスメソッドを実行することができる。(次の例)

例. 通常の形でのメソッドの実行 (先の例の続き)

```
>>> i.im()  Enter  ←インスタンスメソッドの実行
インスタンスメソッドからの出力  ←結果
>>> C.cm()  Enter  ←クラスメソッドの実行
クラスメソッドからの出力  ←結果
```

次に **インスタンス. クラスメソッド (引数)** の形式でクラスメソッドが実行できることを示す。

例. インスタンスに対してクラスメソッドを実行する (先の例の続き)

```
>>> i.cm()  Enter  ←クラスメソッドの実行
クラスメソッドからの出力  ←実行できている
```

次に、**クラス名. インスタンスメソッド (引数)** の形式による実行を試みる。

例. クラス名に対してインスタンスメソッドを実行する試み (先の例の続き)

```
>>> C.im()  Enter  ←インスタンスメソッドの実行を試みるが…
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: C.im() missing 1 required positional argument: 'self'
```

このようにエラーとなるが、**クラス名. インスタンスメソッド (インスタンス, 引数…)** の形式であれば実行できる。

例. クラス名に対してインスタンスメソッドを実行する試み: その2 (先の例の続き)

```
>>> C.im(i)  Enter  ←第1引数にインスタンスを与えると
インスタンスメソッドからの出力  ←実行できる
```

このことは、クラス定義の内部でのメソッドの定義の形式が

```
def メソッド名 (self, 仮引数…):
```

となっていることから理解できる。

2.9.3 setattr, getattr による属性へのアクセス

インスタンス変数やクラス変数にアクセスする場合は基本的にドット表記

オブジェクト名. 属性名

の形式を用いるが、setattr, getattr といった関数によってもアクセスすることができる。

《setattr 関数》

書き方: setattr(オブジェクト名, 属性名, 値)

「オブジェクト名」で示されるオブジェクトの「属性名」(文字列) で示される属性に「値」を設定する。

《getattr 関数》

書き方: getattr(オブジェクト名, 属性名)

「オブジェクト名」で示されるオブジェクトの「属性名」(文字列) で示される属性の値を取得する。

これら関数を使用することでオブジェクトの属性に柔軟な形でアクセスすることができる。例えば、オブジェクト m の属性 x に値 3 を設定する場合、ドット表記を用いて

```
m.x = 3
```

などと記述することができるが、setattr 関数を用いると

```
a = 'x'
setattr( m, a, 3 )
```

という記述で同様のことが実現できる。後者の記述方法を採用することで、上記の変数 a に別の文字列を与えると、setattr の記述の部分を変更することなく m の別の属性に値を設定することが可能となる。

setattr, getattr 関数を用いたサンプルプログラム atracc.py を示す。

プログラム：atracc.py

```
1  # coding: utf-8
2  class MyClass():                # クラス定義
3      cvar = 'クラス変数の値'
4      def __init__(self):
5          self.x = 1
6          self.y = 2
7          self.z = 3
8      def disp(self):
9          print('cvar =', MyClass.cvar)
10         print('x =', self.x, ', y =', self.y, ', z =', self.z)
11
12  m = MyClass()                  # インスタンスの生成
13
14  print('---インスタンスの初期状態---')
15  m.disp()
16
17  # リストに格納されたインスタンス変数名と値を用いた状態の変更
18  alst = ['x','y','z']          # インスタンス変数の名前のリスト
19  vlst = [4,5,6]                # 値のリスト
20  for a,v in zip(alst,vlst):
21      setattr(m,a,v)
22
23  print('---setattrで変更した後の状態 ---')
24  m.disp()
25
26  print('---getattrによるインスタンス変数の参照 ---')
27  for a in alst:
28      print('変数',a,'の値:', getattr(m,a))
29
30  print('---setattrでクラス変数の値を設定 ---')
31  setattr(MyClass,'cvar','新たに設定されたクラス変数の値')
32  m.disp()
```

解説.

x,y,z の3つのインスタンス変数と、クラス変数 cvar を持つ MyClass クラスが定義されている。またこのクラスは、それらの値を表示する disp メソッドを備えている。

12行目で生成されたインスタンス m の属性を 20～21 行目で変更しているが、このとき、18～19 行目で用意した属性名と値のリストを用いて setattr 関数で値の設定を行っている。また、27～28 行目では getattr 関数で m の各種属性を取得して出力している。クラス変数に対しても setattr (31 行目) , getattr 関数でアクセスすることができる。

このプログラムの実行例を次に示す。

実行例.

```
---インスタンスの初期状態---
cvar = クラス変数の値
x = 1 , y = 2 , z = 3
---setattr で変更した後の状態---
cvar = クラス変数の値
x = 4 , y = 5 , z = 6
---getattr によるインスタンス変数の参照---
変数 x の値: 4
変数 y の値: 5
変数 z の値: 6
---setattr でクラス変数の値を設定---
cvar = 新たに設定されたクラス変数の値
x = 4 , y = 5 , z = 6
```

2.9.4 組み込みの演算子や関数との連携

定義したクラスのオブジェクトと、Python に元来備わっている演算子や関数、コンストラクタなどとの間の連携方法について解説する。

2.9.4.1 各種クラスのコンストラクタに対するフック

■ str コンストラクタに対する独自の処理の実装

多くの場合、値やオブジェクトを文字列に変換する場合

`str(値)`

と記述するが、これは `str` クラスのコンストラクタを介したデータ変換である。この方法で、プログラマが独自に定義したクラスのオブジェクトを文字列に変換すると次のようになる。

例. 独自に定義したクラスのインスタンスを `str` で文字列にする

```
>>> class MyClass: Enter    ←クラス定義
...     def __init__(self): Enter
...         self.a = 1 Enter
... Enter    ←クラス定義の記述の終了

>>> x = MyClass() Enter    ← MyClass のインスタンス生成

>>> str(x) Enter    ←それを str で文字列に変換すると…
'<__main__.MyClass object at 0x00000183187A3E88>'    ←このようになる
```

インスタンスを `str` コンストラクタで文字列に変換する際の手順は、当該クラスに `__str__` メソッドとして定義することができる。このことについて次のサンプルプログラム `hook4str01.py` で示す。

プログラム：hook4str01.py

```
1  # coding: utf-8
2  # クラス定義1：__str__の定義なし
3  class C1:
4      def __init__(self):
5          self.a = 1
6          self.b = 2
7          self.c = 3
8
9  x1 = C1()
10 print( 'str(x1):',str(x1) )
11
12 # クラス定義2：__str__の定義あり
13 class C2:
14     def __init__(self):
15         self.a = 1
16         self.b = 2
17         self.c = 3
18     def __str__(self):
19         r = 'a='+str(self.a)+' ,b='+str(self.b)+' ,c='+str(self.c)
20         return r
21
22 x2 = C2()
23 print( 'str(x2):',str(x2) )
```

このプログラムでは `C1`、`C2` の2つのクラスが定義されている。クラス `C1` は `__str__` メソッドが定義されておらず、`C2` にはそれが定義されている。すなわち、クラス `C1` のインスタンスを `str` コンストラクタで文字列に変換すると '`<クラス名 object at …>`' のような文字列となり、`C2` のインスタンスの場合は `__str__` メソッドに記述された形で変換される。

このプログラムをスクリプトとして実行した例を次に示す。

実行例.

```
str(x1): <__main__.C1 object at 0x000002863B5B9CC8>
str(x2): a=1,b=2,c=3
```

このように `__str__` メソッドをクラス内に定義することで、`str` コンストラクタによる変換処理を所望の形式で実行することができる。

参考) クラス内に `__repr__` メソッドを同様の方法で実装することで、`repr` による変換処理を独自の形で実現できる。

■ 各種データ型のコンストラクタに対するフック

先に str コンストラクタに対する独自の処理の実装方法について解説したが、その他のデータ型のコンストラクタに対しても独自の処理を実装することができる。表 25 に、各種コンストラクタに対する独自処理を実装するためのメソッドの対応を記す。

表 25: クラス内に設置するメソッドとそれに対応するコンストラクタ（一部）

クラス内に定義するメソッド	対象となるコンストラクタ	クラス内に定義するメソッド	対象となるコンストラクタ
<code>__int__(self)</code>	<code>int(self)</code>	<code>__float__(self)</code>	<code>float(self)</code>
<code>__complex__(self)</code>	<code>complex(self)</code>	<code>__bool__(self)</code>	<code>bool(self)</code>
<code>__str__(self)</code>	<code>str(self)</code>	<code>__bytes__(self)</code>	<code>bytes(self)</code>

(注) 簡単のため、self の後に続く引数の記述は省略している。

2.9.4.2 算術演算子、比較演算子、各種関数に対するフック

各種の算術演算子、比較演算子、関数に対するフックとなるメソッド名を表 26 に示す。

表 26: クラス内に設置するメソッドとそれに対応する演算子（一部）

クラス内に定義するメソッド	対象となる演算子, 関数	クラス内に定義するメソッド	対象となる演算子, 関数
<code>__add__(self,x)</code>	<code>self + x</code>	<code>__sub__(self,x)</code>	<code>self - x</code>
<code>__mul__(self,x)</code>	<code>self * x</code>	<code>__matmul__(self,x)</code>	<code>self @ x</code> (注)
<code>__truediv__(self,x)</code>	<code>self / x</code>	<code>__floordiv__(self,x)</code>	<code>self // x</code>
<code>__mod__(self,x)</code>	<code>self % x</code>	<code>__divmod__(self,x)</code>	<code>divmod(self,x)</code>
<code>__pow__(self,x)</code>	<code>self ** x</code>	<code>__lshift__(self,x)</code>	<code>self << x</code>
<code>__rshift__(self,x)</code>	<code>self >> x</code>	<code>__and__(self,x)</code>	<code>self & x</code>
<code>__xor__(self,x)</code>	<code>self ^ x</code>	<code>__or__(self,x)</code>	<code>self x</code>
<code>__pos__(self)</code>	<code>+ self</code>	<code>__neg__(self)</code>	<code>- self</code>
<code>__invert__(self)</code>	<code>~ self</code>	<code>__abs__(self)</code>	<code>abs(self)</code>
<code>__eq__(self,x)</code>	<code>self == x</code>	<code>__ne__(self,x)</code>	<code>self != x</code>
<code>__lt__(self,x)</code>	<code>self < x</code>	<code>__le__(self,x)</code>	<code>self <= x</code>
<code>__gt__(self,x)</code>	<code>self > x</code>	<code>__ge__(self,x)</code>	<code>self >= x</code>
<code>__round__(self,n)</code>	<code>round(self,n)</code>	<code>__trunc__(self)</code>	<code>math.trunc(self)</code>
<code>__floor__(self)</code>	<code>math.floor(self)</code>	<code>__ceil__(self)</code>	<code>math.ceil(self)</code>

(注) NumPy ライブラリの行列積で使用

■ インスタンスの比較演算に関する注意

プログラマが独自に定義したクラスのインスタンスに対して == を始めとする比較演算子で比較を行う際には表 26 に示した `__eq__` などのフックを実装すべきである。これに関して例を示して説明する。

例えば次のように定義されたクラス C のインスタンス c1, c2 を作成した場合を考える。

例. クラスの定義とインスタンスの作成

```
>>> class C: pass      ←クラス C の定義
...                      ←クラス定義の終了
>>> c1 = C(); c1.a=1; c1.b=2  ←インスタンス c1 の作成と値の設定
>>> c2 = C(); c2.a=1; c2.b=2  ←インスタンス c2 の作成と値の設定
```

一見すると c1 と c2 は同値であるかのように見えるが == で比較すると次のようになる。

例. == によるインスタンスの比較（先の例の続き）

```
>>> c1 == c2           ← c1 と c2 が同値か判定
False                  ←同値ではない
```

これは c1 と c2 が別々のインスタンスであるので当然の結果であるが、一見して間違い易い。従って、異なるインスタンスの同値性を調べるには当該クラスに `__eq__` を実装しなければならない。（次の例参照）

例. クラスに == による比較を実装する

```
>>> class C: Enter ←クラス定義の開始
...     def __eq__(self,x): Enter ← == による比較演算の実装
...     return True if self.a == x.a and self.b == x.b else False Enter
... Enter ←クラス定義の終了
>>> c1 = C(); c1.a=1; c1.b=2 Enter ←インスタンス c1 の作成と値の設定
>>> c2 = C(); c2.a=1; c2.b=2 Enter ←インスタンス c2 の作成と値の設定
>>> c1 == c2 Enter ←比較の実行
True ←比較結果
```

== 以外の比較演算に関しても同様の考えに基づいてフックを実装すべきである。

■ 二項演算子の項の順序について

表 26 に示した __add__ メソッドを持つ次のようなクラス C1 について考える。

例. __add__ メソッドを持つクラスの定義

```
>>> class C1: Enter
...     def __init__(self,s): Enter
...     self.a = s Enter
...     def __add__(self,x): Enter
...     return str(self.a)+' ':''+str(x) Enter
... Enter
>>> ← Python のプロンプトに戻った
```

このクラス C1 のオブジェクトには

C1 クラスのオブジェクト + 他のオブジェクト

のような演算が可能である。(次の例)

例. '+' 演算の試み (先の例の続き)

```
>>> d1 = C1(' 左側') Enter ← C1 クラスのインスタンス d1 を作成
>>> d1 + ' 右側' Enter ←それに対する '+' の演算
' 左側:右側' ←結果
```

しかし次のような演算ではエラーとなる。

例. 逆の順で '+' を使用する試み (先の例の続き)

```
>>> ' 更に左側' + d1 Enter ←逆順での '+' の演算は
Traceback (most recent call last): ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "C1") to str
```

これは、str クラス (文字列型) の '+' 演算に、クラス C1 のオブジェクトの扱いが定義されていないことによるエラーである。この場合は、クラス C1 にメソッド __radd__ を実装することで解決できる。(次の例)

例. __radd__ メソッドを実装した C1 クラス (先の例の続き)

```
>>> class C1: Enter
...     def __init__(self,s): Enter
...     self.a = s Enter
...     def __add__(self,x): Enter
...     return str(self.a)+' ':''+str(x) Enter
...     def __radd__(self,x): Enter ←上のメソッドがだめな場合に実行される
...     return str(x)+' ':''+str(self.a) Enter ←演算の順序に注意！
... Enter
>>> ← Python のプロンプトに戻った
```

これにより、

他のクラスのオブジェクト + C1 クラスのオブジェクト

が処理できない場合に `__radd__` メソッドが実行される。(次の例)

例. '+' 演算の試み：その2 (先の例の続き)

```
>>> d2 = C1(' 左側')  ← C1 クラスのインスタンス d2 を作成
>>> ' 更に左側' + d2  ← この演算では
' 更に左側:左側'      ← __radd__ が実行される
>>> d2 + ' 右側'      ← もちろんこの演算では
' 左側:右側'          ← __add__ が実行される
```

'+' 演算以外にも、二項演算におけるクラスの順序が原因でエラーとなる場合がある。`__add__` メソッドに対して `__radd__` メソッドがあるように、他の演算子に対しても次のようなメソッド群がある。

```
__radd__,      __rsub__,      __rmul__,      __rmatmul__,      __rtruediv__,
__rfloordiv__, __rmod__,      __rdivmod__,  __rpow__,         __rlshift__,
__rrshift__,   __rand__,      __rxor__,    __ror__
```

詳しくは Python の公式インターネットサイトを参照のこと。

2.9.4.3 累算代入演算子に対するフック

表 27 に累算代入演算子に対するフックとなるメソッドを示す。

表 27: クラス内に設置するメソッドとそれに対応する累算代入演算子

クラス内に定義するメソッド	対象となる演算子	クラス内に定義するメソッド	対象となる演算子
<code>__iadd__(self, x)</code>	<code>self += x</code>	<code>__isub__(self, x)</code>	<code>self -= x</code>
<code>__imul__(self, x)</code>	<code>self *= x</code>	<code>__imatmul__(self, x)</code>	<code>self @= x</code> (注)
<code>__itruediv__(self, x)</code>	<code>self /= x</code>	<code>__ifloordiv__(self, x)</code>	<code>self //= x</code>
<code>__imod__(self, x)</code>	<code>self %= x</code>	<code>__ipow__(self, x)</code>	<code>self **= x</code>
<code>__ilshift__(self, x)</code>	<code>self <<= x</code>	<code>__irshift__(self, x)</code>	<code>self >>= x</code>
<code>__iand__(self, x)</code>	<code>self &= x</code>	<code>__ixor__(self, x)</code>	<code>self ^= x</code>
<code>__ior__(self, x)</code>	<code>self = x</code>		

(注) NumPy ライブラリで使用

2.9.5 インスタンス変数へのアクセスのカスタマイズ

2.9.5.1 存在しないインスタンス変数が参照された際のフック： `__getattr__`

存在しないインスタンス変数を参照するとエラー (`AttributeError`) が起こる。(次の例)

例. 存在しないインスタンス変数の参照

```
>>> class C:  ← クラス定義の記述の開始
...     def __init__(self):  ← クラス定義の記述の開始
...         self.a = 1
...     ← クラス定義の記述の終了
>>> m = C()  ← インスタンスの生成
>>> print( m.a )  ← 存在するインスタンス変数の参照
1
                  ← 値が得られている
>>> print( m.b )  ← 存在しないインスタンス変数の参照を試みると
Traceback (most recent call last):  ← エラーとなる
  File "<stdin>", line 1, in <module>
    print( m.b )
AttributeError: 'C' object has no attribute 'b'
```

クラス定義の中に特殊メソッド `__getattr__` を設置することで、存在しないインスタンス変数への参照があった場合の処理を実装することができる。(次の例)

例. 特殊メソッド__getattr__の設置

```
>>> class C: Enter    ←クラス定義の記述の開始
...     def __init__(self): Enter
...         self.a = 1 Enter
...     def __getattr__(self,attrName): Enter    ←存在しないインスタンス変数への参照を検出するフック
...         print(attrName,'は存在しない属性です') Enter    ←このように出力して
...         return None Enter    ←これを戻り値とする
...     Enter    ←クラス定義の記述の終了
>>> m = C() Enter    ←インスタンスの生成
>>> print( m.a ) Enter    ←存在するインスタンス変数の参照
1 Enter    ←値が得られている
>>> print( m.b ) Enter    ←存在しないインスタンス変数を参照すると
b は存在しない属性です    ←__getattr__メソッドによる出力
None    ←__getattr__メソッドの戻り値
```

この例からわかるように、エラーとならず、参照時の動作が起こり、戻り値を返す。__getattr__の第2引数には、参照が試みられたインスタンス変数の名前が文字列の形で与えられる。

2.9.5.2 インスタンス変数の参照に対するフック：__getattribute__

インスタンス変数の参照は OOP における最も基本的な処理の1つであるが、これに対するフックを特殊メソッド__getattribute__として実装することができる。(次の例)

例. 特殊メソッド__getattribute__、__getattr__の設置

```
>>> class C: Enter    ←クラス定義の記述の開始
...     def __init__(self): Enter
...         self.a = 1 Enter
...     def __getattribute__(self,attrName): Enter    ←インスタンス変数への参照を受けるフック
...         print(attrName,'が参照されました')    ←この出力処理を行い
...         return object.__getattribute__(self,attrName)    ←インスタンス変数の値を返す
...     def __getattr__(self,attrName): Enter    ←存在しないインスタンス変数への参照を検出するフック
...         print(attrName,'は存在しない属性です') Enter    ←このように出力して
...         return None Enter    ←これを戻り値とする
...     Enter    ←クラス定義の記述の終了
>>> m = C() Enter    ←インスタンスの生成
>>> print( m.a ) Enter    ←存在するインスタンス変数の参照
a が参照されました    ←__getattribute__メソッドによる出力
1    ←インスタンス変数の値が__getattribute__メソッドの戻り値として得られる
>>> print( m.b ) Enter    ←存在しないインスタンス変数を参照すると
b が参照されました    ←やはり__getattribute__メソッドが実行される
b は存在しない属性です    ←__getattr__メソッドによる出力
None    ←__getattr__メソッドの戻り値
```

特殊メソッド__getattribute__を設置すると、インスタンス変数への参照があった場合に必ずこれが起動され、このメソッドの戻り値が、インスタンス変数の参照結果の値となる。

__getattribute__の第2引数には、参照されたインスタンス変数の名前が文字列の形で与えられる。また、このメソッドの実装において特に注意すべきこととして、当該インスタンス変数の値の取得に

```
object.__getattribute__(self,attrName)
```

としていることがある。もしも

```
self.__getattribute__(attrName)
```

として参照しようとする、__getattribute__の呼び出しが無限に繰り返されてしまうことが理由である。(試されたい)

2.9.5.3 インスタンス変数への値の代入に対するフック：__setattr__

インスタンス変数への値の代入は OOP における最も基本的な処理の 1 つであるが、これに対するフックを特殊メソッド __setattr__ として実装することができる。(次の例)

例. 特殊メソッド __setattr__ の設置

```
>>> class C: Enter    ←クラス定義の記述の開始
...     def __setattr__(self,attrName,v): Enter    ←インスタンス変数への代入を検出するフック
...     print(' 属性',attrName,' に値',v,' を設定します') Enter    ←代入時にこの出力をして
...     object.__setattr__(self,attrName,v) Enter    ←実際に値を代入する
... Enter    ←クラス定義の記述の終了
>>> m = C() Enter    ←インスタンスの生成
>>> m.a = 1 Enter    ←インスタンス変数 a への代入
属性 a に値 1 を設定します    ←__setattr__メソッドによる出力
>>> m.b = 2 Enter    ←インスタンス変数 b への代入
属性 b に値 2 を設定します    ←__setattr__メソッドによる出力
>>> vars(m) Enter    ← vars 関数150 でインスタンス変数の状態確認
{'a': 1, 'b': 2}    ←値が設定されている
```

特殊メソッド __setattr__ を設置すると、インスタンス変数への値の代入があった場合に必ずこれが起動される。

__setattr__ の第 2 引数には、代入の対象となるインスタンス変数の名前が文字列の形で与えられ、第 3 引数には代入する値が与えられる。また、このメソッドの実装において特に注意すべきこととして、当該インスタンス変数に値を代入する処理として

```
object.__setattr__(self,attrName,v)
```

としていることがある。もしも

```
self.__setattr__(attrName,v)
```

として代入しようとする、__setattr__ の呼び出しが無限に繰り返されてしまうことが理由である。(試されたい)

2.9.6 コンテナのクラスを実装する方法

コンテナの性質を持つクラスを定義する方法について解説する。コンテナはリストや辞書型オブジェクトのように、複数の要素を保持し、スライス '[部分指定の記述]' によって内部にアクセスできるデータ構造であり、次のような機能を持つ。

- 1) len 関数によって要素の個数を返す。(__len__ メソッドの実装)
- 2) in 演算子によって要素の存在を確かめる。(__contains__ メソッドの実装)
- 3) スライスで指定した部分を参照する。(__getitem__ メソッドの実装)
- 4) スライスで指定した部分に値を与える。(__setitem__ メソッドの実装)
- 5) スライスで指定した部分を削除する。(__delitem__ メソッドの実装)

また必要な場合は iter 関数でイテレータにする¹⁵¹ ことができる。

独自のコンテナを実装する例をプログラム ContainerTest01.py に示す。このプログラムはあまり実用的ではないが、上の 1)~5) が実装されていることが簡単に理解できる例である。

プログラム：ContainerTest01.py

```
1  # coding: utf-8
2  # コンテナクラスの定義
3  class MyContainer:
4      def __init__(self,L):          # コンストラクタ
5          self.a = L
6      def __len__(self):             # len関数に対するフック
7          return len(self.a)
8      def __contains__(self,e):      # in演算子に対するフック
9          return e in self.a
```

¹⁵⁰ 「2.9.11.1 クラス変数、インスタンス変数の調査」(p.178) で更に詳しく解説する。

¹⁵¹ これに関しては後の「2.9.7 イテレータのクラスを実装する方法」(p.169) で解説する。

```

10     def __getitem__(self,s):      # スライス指定の参照のためのフック
11         return self.a[s]
12     def __setitem__(self,s,v):    # スライス指定の代入のためのフック
13         self.a[s] = v
14     def __delitem__(self,s):      # スライス指定の削除のためのフック
15         del self.a[s]
16 # 動作確認
17 if __name__ == '__main__':
18     c = MyContainer(['a','b','c','d','e'])
19     print('cの初期状態:',c.a,'長さ:',len(c))
20     print('"d" in c の処理結果:', 'd' in c)
21     print('c[1] の参照:',c[1])
22     print('c[2:4] の参照:',c[2:4])
23     c[3] = 'X'
24     print('c[3] = "X" の処理結果:',c.a)
25     del c[3:]
26     print('del c[3:] の処理結果:',c.a)

```

このプログラムではコンテナの機能を持った MyContainer クラスが定義されており、インスタンス c が作成されている。この c にスライスを付けてアクセスすると、そのスライスが整数値や slice オブジェクト¹⁵² として __getitem__ や __setitem__ といったメソッドの 2 番目の仮引数に渡される。インスタンス c がコンテナとして機能することが次の実行結果からわかる。

実行結果.

```

cの初期状態: ['a', 'b', 'c', 'd', 'e'] 長さ: 5
"d" in c の処理結果: True
c[1] の参照: b
c[2:4] の参照: ['c', 'd']
c[3] = "X" の処理結果: ['a', 'b', 'c', 'X', 'e']
del c[3:] の処理結果: ['a', 'b', 'c']

```

2.9.6.1 マルチスライス (多重スライス)

リストや文字列に対するスライスは、[] の中に単一のスライスオブジェクトを与える形式である。これに対して [s1,s2,...sn] のようにコンマ「,」で区切って複数のスライスオブジェクトを与える形式を採用しているオブジェクト (例えば NumPy ライブラリ¹⁵³ の ndarray オブジェクトなど) もある。このようなスライスの形式は**マルチスライス (多重スライス)** とも呼ばれるもの¹⁵⁴ であり、これを採用するクラスでは、特殊メソッド __getitem__ や __setitem__ において、スライスオブジェクトのタプルを引数に受け取る。これを次のサンプル ContainerTest02.py で例示する。

プログラム: ContainerTest02.py

```

1  # coding: utf-8
2
3  class MyTable:
4      def __init__(self,n,m):      # n行m列の表を作成するコンストラクタ
5          self.a = [ [0]*m for _ in range(n)]      # 初期値の要素は0
6          self.rows = n; self.cols = m
7      def __len__(self):           # len 関数へのフック
8          return self.rows * self.cols      # 全要素数を返す
9      def __contains__(self,e):    # in 演算子へのフック
10         r = False
11         for x in self.a:          # 表内の全要素を調べる
12             if ( r := e in x ): break
13         return r
14     def __getitem__(self,s):      # 表内部の参照
15         if type(s) is tuple:      # マルチスライスの場合
16             return self.a[s[0]][s[1]]
17         else:                     # 単一のスライスの場合
18             return self.a[s]
19     def __setitem__(self,s,v):    # 表の要素への代入
20         if type(s) is tuple:      # マルチスライスの場合
21             self.a[s[0]][s[1]] = v

```

¹⁵² 「2.5.5.2 スライスオブジェクト」 (p.83) を参照のこと。

¹⁵³ 拙書「Python3 ライブラリブック」でも解説しています。

¹⁵⁴ この呼称は PSF が定めたものではないが、本書ではこの呼称を採用する。

```

22         else:                                # 単一のスライスの場合
23             self.a[s] = v
24     def __delitem__(self,s):                    # 表の要素の削除
25         if type(s) is tuple:                    # マルチスライスの場合
26             del self.a[s[0]][s[1]]
27         else:                                    # 単一のスライスの場合
28             del self.a[s]
29     def show(self):                              # 表の内容表示
30         for x in self.a:
31             print( x )

```

次に、ContainerTest02.py を読み込んでその動作を例示する。

例. MyTable クラスのインスタンスの作成

```

>>> from ContainerTest02 import MyTable  Enter   ← MyTable クラスの読み込み
>>> t = MyTable(3,4)  Enter   ← 3行4列の表を作成
>>> t.show()  Enter   ←内容確認
[0, 0, 0, 0]        ←初期値 0 の要素が並ぶ表
[0, 0, 0, 0]
[0, 0, 0, 0]
>>> len(t)  Enter   ←要素数を調べる
12
>>> t[1,2]  Enter   ←インデックス位置の第1行第2列の要素の値を確認
0

```

例. 行列の指定位置で値を設定して確認（先の例の続き）

```

>>> t[1,2] = 3  Enter   ←インデックス位置の第1行第2列に3を設定
>>> t.show()  Enter   ←内容確認
[0, 0, 0, 0]
[0, 0, 3, 0]        ←指定された位置に値が設定されている
[0, 0, 0, 0]
>>> t[1,2]  Enter   ←その位置の値を確認
3

```

例. 行を設定して確認（先の例の続き）

```

>>> t[2] = [4,5,6,7]  Enter   ←インデックス位置の第1行を設定
>>> t.show()  Enter   ←内容確認
[0, 0, 0, 0]
[0, 0, 3, 0]
[4, 5, 6, 7]        ←行が設定されている

```

例. 要素の存在確認（先の例の続き）

```

>>> 6 in t  Enter   ← 6 は t に
True        ←含まれる
>>> 8 in t  Enter   ← 8 は t に
False       ←含まれない

```

実用的なプログラミングにおいては、更に複雑なスライスオブジェクトが与えられることを想定してメソッドを実装すること。

2.9.7 イテレータのクラスを実装する方法

イテレータ¹⁵⁵ の機能を持つクラスを定義する方法について解説する。イテレータとは iter 関数によって作成され、next メソッドによって次々と要素を返し、最後の要素以降の要素を要求すると StopIteration 例外が発生するオブジェクトである。このようなオブジェクトのクラスを定義するには、

- 1) iter 関数が呼ばれた際の処理を実行する `__iter__` メソッド
- 2) next メソッドの処理を実行する `__next__` メソッド

の2つのメソッドを実装する。

次に示す例は、0～10 の整数を次々と返すイテレータオブジェクトのクラス MyIterator を定義するものである。

¹⁵⁵詳しくは「2.6.1.8 イテレータ」(p.92) を参照のこと。

例. イテレータオブジェクトのクラス定義

```
>>> class MyIterator: Enter ←クラス定義
...     def __init__(self): Enter ←コンストラクタ
...         self.n = 0 Enter
...     def __iter__(self): Enter ← iter 関数に対するフック
...         return self Enter
...     def __next__(self): Enter ← next メソッドに対するフック
...         r = self.n Enter
...         self.n += 1 Enter
...         if r > 10: Enter ←最終要素を越えたことの判定
...             raise StopIteration() Enter ← StopIteration 例外を発生させる
...         return r Enter
... Enter ←クラス定義の記述の終了
>>> ←対話モードのプロンプトに戻った
```

`__iter__`, `__next__` の各メソッドを実装している。この例では、`iter` 関数の呼び出しに対しては特に行うことが無く、自身のオブジェクト (`self`) をそのまま返している。また、`next` メソッドの実行によって返す値は「10」までとしているので、それ以降に要素が要求された場合に `raise` 文¹⁵⁶ で `StopIteration` 例外を発生させている。

実際にこのクラスのオブジェクトを生成してイテレータとして使用する例を次に示す。

例. MyIterator クラスのインスタンスをイテレータとして使用する (先の例の続き)

```
>>> It = MyIterator() Enter ←インスタンスの生成
>>> for x in It: Enter ←次々と要素を取り出し
...     print(x,end=' ') Enter ←それを出力 (改行なし) して
... else: print('') Enter ←最後に改行する
... Enter ← for 文の記述の終了
0 1 2 3 4 5 6 7 8 9 10 ←実行結果の出力
```

2.9.8 カプセル化

オブジェクト指向プログラミングの基本的な考え方においては、インスタンス内部の変数 (インスタンス変数、プロパティ、属性) にインスタンス外部から直接的にアクセスするのは良いことではないとされる。オブジェクトはデータ構造とメソッドを組にした構造物であり、内部構造には基本的には外部からアクセスできないように隠蔽しておき、アクセサと呼ばれるメソッドを介してのみオブジェクトの内部構造にアクセスすべきであるというのがカプセル化の考え方である。ここではカプセル化に関連する事柄について説明する。

2.9.8.1 変数の隠蔽

アンダースコア 2 つ `__` で始まるインスタンス変数、クラス変数は隠蔽され、オブジェクトの外部からはアクセスできない¹⁵⁷。このことを次の例で示す。

例. クラス内の変数の隠蔽

```
>>> class C: Enter ←クラス定義の開始
...     def __init__(self): Enter ←コンストラクタ
...         self.a = 2 Enter ←外部からアクセスできる変数
...         self.__b = 3 Enter ←隠蔽される変数
...     def show(self): Enter ←内部状態を表示するメソッド
...         print(' a:',self.a)
...         print(' __b:',self.__b) ←内部からは __b にアクセスできる
... Enter ←クラス定義の終了
>>> ←対話モードのプロンプトに戻った
```

¹⁵⁶詳しくは「4.19 例外 (エラー) の処理」(p.337) で解説する。

¹⁵⁷実際には、変数名が装飾されて別の名前になっているだけである。

この例で定義されたクラス C のインスタンスは

- (1) 外部からアクセスできる変数 a
- (2) 隠蔽される変数 __b

を持つ。(1)(2) それぞれの変数にアクセスする例を次に示す。

例. インスタンス変数へのアクセス（先の例の続き）

```
>>> x = C()      Enter      ←クラス C のインスタンス x の作成
>>> x.a          Enter      ←インスタンス変数 a の参照
2                  ←参照できた
>>> x.__b        Enter      ←インスタンス変数 __b の参照を試みると…
Traceback (most recent call last):      ←そのようなものが存在しない旨のエラーとなる
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '__b'
>>> x.show()     Enter      ← show メソッドを介してアクセスすると
a: 2
__b: 3            ←参照できている
```

変数 __b は外部からは直接アクセスできないことがわかる。また、この例では show メソッドがアクセサとしての役割を果たしている。

▲注意▲

変数の前後にアンダースコア 2 つを付けると隠蔽されないので注意すること。

■ 隠蔽された変数に関する考察

アンダースコア 2 つ '__' で始まる変数に対して、クラス定義の外部で値の代入を試みる。

例. インスタンスの変数 __b に値を代入する試み（先の例の続き）

```
>>> x.__b = 99    Enter      ←代入を試みる
>>> x.__b         Enter      ←代入後の値を確認
99                  ←値が設定できている
>>> x.show()      Enter      ← show メソッドによる確認
a: 2
__b: 3            ←最初のまま変化していない
```

この例からわかるように、クラス定義の記述の中部での変数 __b と、外部での変数 __b は別のものであることがわかる。更に、vars 関数¹⁵⁸を用いて、インスタンス x が保持する変数を調べる処理を例示する。

例. インスタンスが保持する変数を調べる（先の例の続き）

```
>>> vars(x)       Enter      ← x が持つ変数を調査
{'a': 2, '_C__b': 3, '__b': 99} ←変数とその値が辞書として得られる
```

この例からわかるように、クラス定義の記述の内部のアンダースコア 2 つ '__' で始まる変数は、「_クラス名」で装飾されている。

2.9.8.2 アクセサ（ゲッター, セッター）

先の例では通常のメソッドをアクセサとしたが、ここでは Python での標準的なアクセサの扱いを示す。

■ ゲッター (getter)

隠蔽された変数の値を参照するためのアクセサをゲッター (getter) と呼ぶ。ゲッターの実装方法について例を挙げて説明する。次の例は、人の身長と体重を保持し、BMI を算出するメソッドを持つクラスを定義するものである。

¹⁵⁸ 「2.9.11.1 クラス変数、インスタンス変数の調査」(p.178) で解説する。

例. クラス Health の定義

```
>>> class Health: Enter ←クラス定義の開始
...     def __init__(self,h,w): Enter ←コンストラクタ
...         self.__height = h Enter ←隠蔽される変数
...         self.__weight = w Enter ←隠蔽される変数
...     @property Enter ←デコレータ：参照用アクセサ（ゲッター）
...     def height(self): Enter ←アクセサ：身長
...         return self.__height Enter
...     @property Enter ←デコレータ：参照用アクセサ（ゲッター）
...     def weight(self): Enter ←アクセサ：体重
...         return self.__weight Enter
...     def bmi(self): Enter ←BMI を求めるメソッド
...         return self.weight / self.height**2 Enter
... Enter ←クラス定義の終了
>>> ←対話モードのプロンプトに戻った
```

このクラスのインスタンスは身長 (m) と体重 (kg) を与えて作成する。height, weight メソッドの宣言の直前にデコレータ「@property」が記述されている。このデコレータにより、height, weight メソッドは引数を付けずに実行することができる。(次の例)

例. Health クラスのインスタンスの扱い (先の例の続き)

```
>>> taro = Health( 1.7, 65 ) Enter ←クラス Health のインスタンス taro の作成
>>> taro.weight Enter ←引数を付けずに実行
65 ←値が得られている
>>> taro.height Enter ←引数を付けずに実行
1.7 ←値が得られている
>>> taro.bmi() Enter ←メソッド bmi の実行には引数のための括弧の記述が必要
22.49134948096886 ←BMI 値
```

height, weight はあたかもプロパティであるかのように扱える。Python では、インスタンス内部の変数を参照するためのアクセサ（ゲッター）をこのような形で実装するのが標準的である。

■ セッター (setter)

隠蔽された変数に値を設定するためのアクセサをセッター (setter) と呼ぶ。基本的には、セッターはゲッターと対にして実装するものであり、ゲッターの定義の後に同名のセッターの実装を記述する。セッターとするメソッドの記述の直前にはデコレータ「@セッターの名前.setter」を記述する。(プロパティ名としてゲッターと同じ名前とする)

セッターの実装方法について例を挙げて説明する。次の例は、隠蔽された変数 __price を持つクラス D を定義するものである。

例. ゲッターとセッターを持つクラス D

```
>>> class D: Enter ←クラス定義の開始
...     def __init__(self): Enter ←コンストラクタ
...         self.__price = 0 Enter ←隠蔽される変数
...     @property Enter ←デコレータ：参照用アクセサ（ゲッター）
...     def price(self): Enter ←ゲッターの定義
...         return self.__price Enter
...     @price.setter Enter ←デコレータ：設定用アクセサ（セッター）
...     def price(self,p): Enter ←セッターの定義（ゲッターと同じメソッド名）
...         self.__price = p Enter
...         return self.__price Enter
...     def show(self): Enter ←内部状態を表示するメソッド
...         print('price:',self.price) Enter
... Enter ←クラス定義の終了
>>> ←対話モードのプロンプトに戻った
```

このクラスのインスタンスに対してセッター、ゲッターを使用する例を次に示す。

例. クラス D のインスタンスの扱い（先の例の続き）

```
>>> apple = D()      Enter      ←クラス D のインスタンス apple の作成
>>> apple.show()     Enter      ←内部状態の表示
price: 0              ←値が表示された
>>> apple.price = 100 Enter      ←セッターを介して内部の変数に値を設定
>>> apple.show()     Enter      ←内部状態の表示
price: 100            ←値が表示された
>>> apple.price      Enter      ←ゲッターを介して内部の変数を参照
100                   ←参照された値
```

セッターのメソッドもゲッターと同一のメソッド名として定義している。セッターも引数を付けずに実行することができる。結果として、インスタンス apple の隠蔽された変数 `__price` に対して、`apple.price` という記述で設定と参照の両方が実行できていることがわかる。

2.9.9 メソッドのオーバーライドと `super()`

基底クラスで定義したメソッドと同じ名前のメソッドを拡張クラス側で定義すると、拡張クラス側のメソッドが基底クラスのメソッドをオーバーライドする。すなわち、拡張クラス側で実行されるメソッドは、そのクラスで定義されたものであり、基底クラス側のメソッドは無効になる。ただし `super` を用いると基底クラス側のメソッドを呼び出すことができる。このことを次の `overrideTest1.py` を例に挙げて解説する。

プログラム：`overrideTest1.py`

```
1 class A:
2     def method1(self):
3         print('method1 of class A')
4
5 class B(A):
6     def method1(self):
7         super().method1()
8         print('method1 of class B')
9
10 class C(B):
11     def method1(self):
12         super().method1()
13         print('method1 of class C')
14
15 print('クラス A のインスタンスに対する method1() の実行。')
16 a = A()
17 a.method1()
18
19 print('クラス B のインスタンスに対する method1() の実行。')
20 b = B()
21 b.method1()
22
23 print('クラス C のインスタンスに対する method1() の実行。')
24 c = C()
25 c.method1()
```

このプログラムでは、クラスが順に `A → B → C` と派生しており、全てのクラスで同一の名前のメソッド `method1` が定義されている。各クラスのインスタンスにこのメソッドを実行すると、当該クラスの `method1` が実行される。しかし、`b`、`c` に対して `method1` を実行すると、その定義に記述された

```
super().method1()
```

によって、1つ上の基底クラスの `method1` も実行される。「`super()`」は **super オブジェクト**であり、ここでは、当該インスタンス (`self`) を 便宜的に1つ上の基底クラスのものとして解釈できる。詳しくは後の「2.9.9.1 `super` オブジェクト」(p.174)で解説する。

`super()` によって、オーバーライドされて無効になった基底クラスのメソッドを実行することができ、`overrideTest1.py` を実行すると次のような出力が得られる。

実行例.

クラス A のインスタンスに対する method1() の実行.

method1 of class A

クラス B のインスタンスに対する method1() の実行.

method1 of class A

method1 of class B

クラス C のインスタンスに対する method1() の実行.

method1 of class A

method1 of class B

method1 of class C

←親クラスの method1 を遡及実行

←自クラスの method1 による出力

←先祖クラスの method1 を遡及実行

←親クラスの method1 を遡及実行

←自クラスの method1 による出力

クラス B のインスタンスに method1 を実行すると、クラス A に定義された method1 が実行されていることがわかる。また、クラス C のインスタンス c に対して method1 を実行すると、クラス A の method1 まで遡及して実行されていることがわかる。

super() はクラスメソッドにおいても使用できる。このことを次の overrideTest2.py で示す。

プログラム：overrideTest2.py

```
1 class A:
2     @classmethod
3     def method1(cls):
4         print('method1 of class A')
5
6 class B(A):
7     @classmethod
8     def method1(cls):
9         super().method1()
10        print('method1 of class B')
11
12 class C(B):
13     @classmethod
14     def method1(cls):
15         super().method1()
16         print('method1 of class C')
17
18 print('クラス A に対する method1() の実行。')
19 A.method1()
20
21 print('クラス B に対する method1() の実行。')
22 B.method1()
23
24 print('クラス C に対する method1() の実行。')
25 C.method1()
```

このプログラムでは、先の overrideTest1.py と類似の考え方で上位のクラスのクラスメソッド method1 を遡及して実行する。overrideTest2.py を実行すると次のような出力が得られる。

実行例.

クラス A に対する method1() の実行.

method1 of class A

クラス B に対する method1() の実行.

method1 of class A

method1 of class B

クラス C に対する method1() の実行.

method1 of class A

method1 of class B

method1 of class C

2.9.9.1 super オブジェクト

先に super オブジェクトのことを「当該インスタンス (self) を便宜的に 1 つ上の基底クラスのものとして解釈できる」と解説したが、正確には、祖先のクラスを参照するための特殊なオブジェクトである。実際に super() が返す値のクラスは super である。これは次のプログラム overrideTest3.py で確認できる。

プログラム：overrideTest3.py

```
1 class A:
2     def method1(self):
3         print('method1 of class A')
```

```

4
5 class B(A):
6     def method1(self):
7         print('type(super()) :', type(super()))
8         super().method1()
9         print('method1 of class B')
10
11 b = B()
12 b.method1()

```

このプログラムでは、クラス B のメソッド method1 の中で

```
type( super() )
```

として super オブジェクトのクラスを調べ、それを出力する。このプログラムを実行すると次のような出力が得られる。

実行例.

```

type(super()) : <class 'super'>          ← super オブジェクトのクラス (型)
method1 of class A
method1 of class B

```

このプログラムからもわかるように、「super()」はクラス super のコンストラクタである。またこのコンストラクタは、実行される状況から適切に基底クラスを参照する。また、クラス super のコンストラクタには次のように引数を与えることもできる。

書き方： super(クラス, インスタンス)

「クラス」「インスタンス」の情報から、「クラス」の基底クラスを参照する super オブジェクトを作成して返す。これに関して次のようなプログラム overrideTest4.py を示して解説する。

プログラム：overrideTest4.py

```

1 class A:
2     def method1(self):
3         print('method1 of class A')
4
5 class B(A):
6     def method1(self):
7         super().method1()
8         print('method1 of class B')
9
10 class C(B):
11     def method2(self):
12         super(B,self).method1()    # クラス B の親 (つまりクラス A) を参照
13         print('method2 of class C')
14
15 print('クラス C のインスタンスに対する method1 () の実行. ')
16 c = C()
17 c.method1()
18
19 print('クラス C のインスタンスに対する method2 () の実行. ')
20 c.method2()

```

このプログラムでは、クラス C のメソッド method2 の中に

```
super(B,self)
```

という記述がある。これは「クラス B の基底クラスのインスタンス」と解釈することができ、結果としてクラス A のインスタンスと見なして method1 を実行している。このプログラムを実行すると次のような出力が得られる。

実行例.

```

クラス C のインスタンスに対する method1 () の実行.
method1 of class A          ←これはクラス B から継承された method1
method1 of class B          による出力
クラス C のインスタンスに対する method2 () の実行.
method1 of class A          ←クラス A の method1 を直接的に実行
method2 of class C

```

実行結果から、クラス C 内からクラス B の method1 を実行することなく、クラス A の method1 を実行していることがわかる。

2.9.10 インスタンス生成に関する制御：__new__

Python では、クラスのインスタンスが生成された直後に `__init__` メソッドが実行され、当該インスタンスの初期化処理が行われるが、インスタンスの生成処理自体は特殊メソッド `__new__` によって行われる。通常の場合、インスタンス生成の直前に自動的に `__new__` メソッドが実行され、この処理を意識することはない。また、`__new__` メソッドは、クラス階層の最上位のクラスである `object` クラスのものを起点として、全てのクラスはこれを継承している。

`__new__` メソッドはユーザが定義するクラスでオーバーライドすることも可能であり、インスタンス生成に関するユーザ独自の制御が可能である。次に示す例は、コンストラクタが常に同じオブジェクトを返すクラスの定義である。

例. クラス Identical

```
>>> class Identical:  ←クラス定義の開始
...     __inst = ('Identical',)  ←Enter
...     def __new__(cls):  ←Enter
...         return cls.__inst  ←Enter
...  ←Enter  ←クラス定義の記述の終了
```

このように `__new__` メソッドの最初の仮引数には当該クラスを受け取る。ただし、通常のクラスメソッドのようにデコレータ `@classmethod` を伴わない。上の例では、このクラスのコンストラクタが呼び出されると、クラス変数である `s` の値が無条件に返される。すなわち、オブジェクトの新たな生成は行わず、常にこの変数の値が返される。(次の例)

例. MySingleton コンストラクタで得られるオブジェクト (先の例の続き)

```
>>> x = Identical()  ←コンストラクタでオブジェクトを取得 (1)
>>> x  ←Enter  ←内容確認
('Identical',)
>>> y = Identical()  ←コンストラクタでオブジェクトを取得 (2)
>>> y  ←Enter  ←内容確認
('Identical',)
>>> x is y  ←Enter  ←x と y は同一のオブジェクト
True  ←である。
```

コンストラクタで `x` と `y` にオブジェクトを得ているが、それらが同一のオブジェクトであることがわかる。もちろんそれらの値は、事後に作った `('MySingleton',)` とは別のものである。(次の例)

例. `x`, `y` の同一性の調査 (先の例の続き)

```
>>> z = ('Identical',)  ←Enter  ←x, y と同じ内容の別オブジェクト
>>> x is z  ←Enter  ←x と z は同一のオブジェクトか?
False  ←違う
>>> print(id(x),id(y),id(z),sep='¥n')  ←Enter  ←x, y, z の識別値を調べる
2360142629040  ←x と
2360142629040  ←y は同じだが,
2360142630672  ←z は違う
```

以上のことから `Identical` は、コンストラクタが常に同じオブジェクトを返すクラスであることがわかる。(上の例の識別値は実行時に自動的に決められる)

注意) 上に示した `Identical` クラスの設計パターンは実際のプログラミングにおいては推奨されない。

`__new__` メソッドはインスタンスを生成するためのものであり、次に解説するように、基底クラスの `__new__` メソッドを呼び出す形で設計するべきである。

2.9.10.1 新規インスタンス生成の仕組み

Python で新規クラスを定義する場合、基底クラス (スーパークラス) の `__new__` メソッドが暗黙のうちに継承される。これにより、当該新規クラスのコンストラクタを実行すると、そのクラスの `__new__` メソッドが実行される。そしてその中に記述されている

```
return super().__new__(cls, 引数並び…)
```

が実行され、新規オブジェクトが生成されて返される。この処理は最上位の `object` クラスまで連鎖的に実行される。

すなわち、object クラスの `__new__` が実際に新たなオブジェクトを生成し、それがその拡張クラスのインスタスとなる。

「引数並び」にはコンストラクタに与えた引数が渡される。

2.9.10.2 シングルトン

インスタンスを1回だけ生成し、以降はその同じインスタンスを返すようなクラスをシングルトンであるという。シングルトンクラスの例を次に示す。

例. シングルトンクラス MySingleton

```
>>> class MySingleton: Enter    ←クラス定義の開始
...     __inst = None Enter
...     def __new__(cls,*a,**ka): Enter
...         if cls.__inst: Enter
...             return cls.__inst Enter
...         else: Enter
...             cls.__inst = super().__new__(cls) Enter
...             return cls.__inst Enter
...     def __init__(self,v): Enter
...         self.value = v Enter
... Enter    ←クラス定義の記述の終了
```

このクラス MySingleton は、隠蔽されたクラス変数 `__inst` を持ち、初期状態では None となっている。このクラスでは、コンストラクタが最初に呼び出された際に、`super().__new__(cls)` が実行されて新規オブジェクトが1つ生成され、続いて `__init__` が呼び出されて初期化処理が施された後、それがこのクラスのインスタンスとして返される。しかし、2回目以降のコンストラクタの呼び出しでは、初回に生成されたインスタンスに対して `__init__` を実行して返す。

MySingleton クラスのインスタンス生成の例を次に示す。

例. MySingleton のコンストラクタ（先の例の続き）

```
>>> x = MySingleton(7) Enter    ←初回のインスタンス生成
>>> x.value Enter    ←値（value 属性）の確認
7                                ←コンストラクタの引数に与えた値
>>> y = MySingleton(9) Enter    ←2回目のインスタンス生成
>>> y.value Enter    ←値（value 属性）の確認
9                                ←コンストラクタの引数に与えた値
```

この例ではインスタンスが x, y にえられているが、これらが同一のインスタンス（初回生成時のもの）であることを確認する。（次の例）

例. x, y が同一のものであることの確認（先の例の続き）

```
>>> x.value = 11 Enter    ←初回生成時のインスタンスの value 属性に値を設定
>>> y.value Enter    ←2回目生成時のインスタンスの value 属性の値を確認
11                                ←初回生成時のインスタンスのものと同一
>>> print(id(x),id(y),sep='¥n') Enter    ← x, y の識別値を調べる
2208385239760                    ←どちらも同じ
2208385239760                    ←識別値である
```

需要) 上に示したクラス設計のパターンは、データベース接続、ファイルアクセス、通信といった局面で、同一の処理対象に継続的に処理を実行する場合に応用できる。

2.9.11 オブジェクト指向プログラミングに関するその他の事柄

2.9.11.1 クラス変数, インスタンス変数の調査

`vars` 関数を使用すると, クラスが持つクラス変数を調べることができる. 同様にインスタンスが持つインスタンス変数を調べることもできる.

例. サンプル用のクラス定義

```
>>> class C:
...     x = 1
...     y = 2
...     def __init__(self):
...         self.a = 3
...         self.b = 4
...
>>> ← Python のプロンプトに戻った
```

これは, クラス変数 `x`, `y` を, インスタンス変数 `a`, `b` を持つクラスの定義である. このクラスのインスタンスを作成し, それらを調べる例を次に示す.

例. インスタンス変数を調べる (先の例の続き)

```
>>> x = C()
>>> vars(x)
{'a': 3, 'b': 4}
```

← クラス `C` のインスタンス `x` を作成
← `x` のインスタンス変数を調べる
← 辞書オブジェクトの形で得られる

このように結果が辞書オブジェクトの形で得られる.

例. クラス変数を調べる (先の例の続き)

```
>>> vars(C)
mappingproxy({'__module__': '__main__', 'x':1, 'y':2,
'__init__': <function C.__init__ at 0x0000020FF4B677E0>,
'__dict__': <attribute '__dict__' of 'C' objects>,
'__weakref__': <attribute '__weakref__' of 'C' objects>, '__doc__': None})
```

← クラス `C` のクラス変数を調べる
← `x`, `y` が含まれていることがわかる

クラス変数を `vars` 関数で調べると, `mappingproxy` オブジェクトの形 (参照専用の辞書) で結果が得られる.

注意) 上記 `<function C.__init__ at 0x0000020FF4B677E0>` のアドレスは実行時に自動的に決定される.

■ 属性を管理する `__dict__` 属性

クラスやそのインスタンスが持つ変数は `__dict__` 属性で管理されており, `vars` 関数はその内容を返す.

例. `__dict__` 属性 (先の例の続き)

```
>>> x.__dict__
{'a': 3, 'b': 4}
>>> C.__dict__
mappingproxy({'__module__': '__main__', 'x':1, 'y':2,
'__init__': <function C.__init__ at 0x0000020FF4B677E0>,
'__dict__': <attribute '__dict__' of 'C' objects>,
'__weakref__': <attribute '__weakref__' of 'C' objects>, '__doc__': None})
```

← `x` のインスタンス変数を調べる
← クラス `C` のクラス変数を調べる
← `x`, `y` が含まれていることがわかる

`vars` 関数は対象オブジェクトの `__dict__` 属性を使用するため, この属性を持たないオブジェクトを引数に与えると `TypeError` となるので注意すること.

2.9.11.2 クラスの継承関係の調査

あるクラスが別のクラスの派生クラス (拡張クラス, サブクラス) となっているかどうか (継承関係) を調べるには `issubclass` 関数を使用する. 以下に例を挙げて説明する.

例. 継承関係のあるクラス定義

```
>>> class A: pass Enter ←クラス A の定義（中身は無し）
... Enter ←定義の終了
>>> class B(A): pass Enter ←クラス A を継承するクラス B の定義（中身は無し）
... Enter ←定義の終了
>>> class C(B): pass Enter ←クラス B を継承するクラス C の定義（中身は無し）
... Enter ←定義の終了
```

これで $A \rightarrow B \rightarrow C$ の継承関係ができた。次にそれを調べる。

例. `issubclass` による継承関係の調査（先の例の続き）

```
>>> issubclass(C,B) Enter ← C は B のサブクラスか？
True ←真：C は B のサブクラスである
>>> issubclass(C,A) Enter ← C は A のサブクラスか？
True ←真：C は A のサブクラスである
>>> issubclass(A,B) Enter ← A は B のサブクラスか？
False ←偽：A は B のサブクラスではない
```

このように、直接の継承関係だけでなく、直系¹⁵⁹ である場合も検査可能である。

2.9.11.3 インスタンスのクラスの調査

インスタンスのクラスを調べるには `type` 関数を用いる。また、あるインスタンスがあるクラスに属しているかどうかを検査するには `isinstance` を用いる。

例. クラスの調査（先の例の続き）

```
>>> x = C() Enter ←クラス C のインスタンス x を生成
>>> type(x) Enter ← x のクラスを調べる
<class '__main__.C'> ←クラス C のインスタンスである
>>> isinstance(x,C) Enter ← x はクラス C のインスタンスか？
True ←判定結果
>>> isinstance(x,A) Enter ← x はクラス A のインスタンスか？
True ←判定結果
```

このように `isinstance` は、属するクラスのスーパークラス（上位のクラス）の場合も `True` を返す。

インスタンスが属するクラスは、そのインスタンスの `__class__` 属性からも参照できる。

例. `__class__` 属性の参照（先の例の続き）

```
>>> x.__class__ Enter ← x のクラスを調べる
<class '__main__.C'> ← x のクラスは C である
```

課題. 先のクラス定義において `C.__class__` の値はどのようなものであるか調べよ。また、それが意味することについて考察せよ。

2.9.11.4 属性の調査（プロパティの調査）

`dir` 関数を使用すると、オブジェクトの属性（プロパティ、メソッド）のリストを取得できる。

例. 文字列クラス `str` の属性をリストとして取得する

```
>>> dir(str) Enter ← str クラスの調査
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 (途中省略)
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

参考) `dir` 関数はインスタンスの属性を調べることもできる。

¹⁵⁹ 「サブクラスのサブクラスの…」と継承関係の系列上にあるという意味。

オブジェクトが特定の属性（プロパティ）を持つかどうかは `hasattr` 関数で調べることができる。

例. 文字列クラス `str` の属性の調査

```
>>> hasattr( str, 'split' )  Enter      ← str クラスが 'split' を持つか
True                        ← 'split' という名の属性を持つ
>>> hasattr( str, 'qwert' )  Enter      ← str クラスが 'qwert' を持つか
False                       ← 'qwert' という名の属性を持たない
```

2.9.11.5 多重継承におけるメソッドの優先順位

あるクラスが他の複数のクラスを基底クラスとして多重継承する場合、それら基底クラスに同じ名前のインスタンスメソッドがある場合、どの基底クラスのインスタンスメソッドが実行されるかが問題となる。例えば次のようなクラス定義について考える。

例. クラス A,B を多重継承するクラス C

```
>>> class A:  Enter
...     def __init__(self):  Enter
...         self.a = 'クラス A の属性 a の値'  Enter
...     def imethod(self):  Enter
...         print('クラス A のインスタンスメソッドによる出力')  Enter
...  Enter
>>> class B:  Enter
...     def __init__(self):  Enter
...         self.b = 'クラス B の属性 b の値'  Enter
...     def imethod(self):  Enter
...         print('クラス B のインスタンスメソッドによる出力')  Enter
...  Enter
>>> class C(A,B):  Enter
...     def __init__(self):  Enter
...         self.c = 'クラス C の属性 c の値'  Enter
...  Enter
>>>  ← Python のプロンプトに戻った
```

このような継承関係では、クラス A, B が同じ名前のインスタンスメソッド `imethod` を持っており、クラス C のインスタンスに対して `imethod` を実行すると、A, B どちらのものが実行されるかが問題となる。実際にクラス C のインスタンスを作成して実行する例を次に示す。

例. クラス C のインスタンスに対して `imethod` を実行する（先の例の続き）

```
>>> m = C()  Enter      ← インスタンス m を作成して
>>> m.imethod()  Enter   ← インスタンスメソッド imethod を実行すると
                        クラス A のインスタンスメソッドによる出力 ← クラス A のものが実行される
```

これは、クラス C を定義する際に `class C(A,B):` としたことによりクラス A が優先されたことによる。多重継承で定義されたクラスにおいてメソッドの優先順を調べるには、当該クラスに対して `mro` メソッドを使用すると良い。

例. クラス間の優先順位を調べる（先の例の続き）

```
>>> C.mro()  Enter      ← クラス C が継承しているクラスの間の優先順位を調べる
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

クラスの参照順が $C \rightarrow A \rightarrow B \rightarrow (\text{その他})$ となっていることがわかる。

上のような状況で、インスタンス `m` にクラス B のインスタンスメソッド `imethod` を強制的に実行させるには次のような方法がある。

例. クラス B のインスタンスメソッドを実行する（先の例の続き）

```
>>> B.imethod(m)  Enter      ← クラス B 配下の imethod であることを明確に記述する
                        (クラス B のインスタンスメソッドによる出力)
```

2.9.11.6 静的メソッド（スタティックメソッド）

クラス内に定義されたスタティックメソッドは通常関数のように振る舞う。これに関して例を挙げて説明する。

例. 静的メソッドを持つクラスの定義

```
>>> class C: Enter
...     @staticmethod Enter ←静的メソッドの宣言
...     def smethod(x): Enter ←仮引数に self や cls が無い
...         print('x :',x) Enter
...         return 2*x Enter
...     def __init__(self): Enter
...         self.a = 5 Enter
...     def imethod(self,x): Enter
...         print('self :',self,'x :',x) Enter
...         return x*self.a Enter
... Enter
>>> ← Python のプロンプトに戻った
```

このクラス C のインスタンスメソッドと静的メソッドの動作を比較する。

例. インスタンスメソッドの動作と静的メソッドの動作（先の例の続き）

```
>>> m = C() Enter ←クラス C のインスタンス m を作成
>>> m.imethod(3) Enter ← m に対してインスタンスメソッドを実行
self : <__main__.C object at 0x000002C9E92BF250> x : 3 ←引数 self が自身を受け取っている
15 ←戻り値
>>> C.smethod(3) Enter ←静的メソッドを実行
x : 3 ←self や cls などを受け取らずに実行している
6 ←戻り値
```

この例の smethod はインスタンス m に対しても同様に実行することができる。

このように静的メソッドは、self や cls といった仮引数を取らずに実装できるので、通常関数のように扱うことができる。

課題. 静的メソッドの用途について考察せよ。

2.9.11.7 クラスの定義の削除

クラス定義は del で削除することができる。ただしこれはあまり推奨されないので注意すること。

定義したクラスを削除する例を示す。

例. クラス C の定義とインスタンスの生成

```
>>> class C: Enter ←クラス C の定義の記述の開始
...     def __init__(self): Enter
...         self.a = 999 Enter
... Enter ←クラス定義の記述の終了
>>> x = C() Enter ←インスタンス x の生成
>>> x.a Enter ←インスタンス変数の参照
999 ←値が参照できている
```

次に、このクラス C を削除する。

例. クラス C の削除 (先の例の続き)

```
>>> del C      [Enter]      ←クラス C の削除
>>> y = C()    [Enter]      ←インスタンス生成を試みると…
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    y = C()
      ^
NameError: name 'C' is not defined      ← C というクラス名は既に存在しない
>>> x.a        [Enter]      ←しかしこの段階ではインスタンス x は残留しており
999              ←インスタンス変数の参照ができています
>>> type(x)     [Enter]      ←インスタンス x のクラスを
<class '__main__.C'>      ←調べることができる
```

このように del でクラス名を削除することができるが、既存のインスタンスは存在し続ける。またこの例では、クラス名 C が削除されただけであり、クラス定義自体はインスタンス x が参照しているため、クラス名 C が無効になったあともその実体はまだ存在している。(次の例)

例. まだ存在するクラス定義の実体 (先の例の続き)

```
>>> y = type(x)() [Enter]      ←残留しているクラス定義からインスタンスを生成
>>> y.a          [Enter]      ←インスタンス変数の参照
999              ←値が参照できている
>>> type(y)       [Enter]      ←クラスを調べることもできる
<class '__main__.C'>      ←残留しているクラス定義
```

削除されたクラスを参照しているインスタンスが完全に消滅すると、そのクラスの定義も消滅する。

例. クラス C のインスタンスを全て削除 (先の例の続き)

```
>>> del x, y     [Enter]      ←これによってクラス C の定義の実体も削除される
```

全ての参照がなくなることによって、ガベージコレクションのタイミングで型オブジェクトも自動的に解放される。

2.10 データ構造に則したプログラミング

Python にはデータ列の要素に対して一斉に処理を適用する機能が提供されている。例えば 2 倍した値を返す関数 `dbl` が次のように定義されているとする。

例. 2 倍する関数 `dbl` の定義

```
def dbl(n): return 2*n
```

通常は下記のように、これを 1 つの値に対して実行する。

例. 通常の形式での関数の実行（先の例の続き）

```
>>> dbl(3)  Enter      ←関数の評価
6          ←評価結果
```

この評価方法とは別に、**map 関数**を使用する方法がある。map 関数を使うと、データ列の全ての要素に対して評価を適用することができる。このことを例を挙げて説明する。

例. map 関数による一斉評価（第一段階）（先の例の続き）

```
>>> res = map( dbl, [1,2,3] )  Enter
```

これにより、関数の評価をリストの全要素に対して一斉に行うことができる。ただしこの段階では「関数の一斉評価のための式」が `res` に生成されただけであり、`res` の内容を確認すると次のようになっている。

例. map 関数の実行結果（先の例の続き）

```
>>> res  Enter      ←内容の確認
<map object at 0x000002BE46760DD8> ←map オブジェクトが格納されている
```

この `res` をリストとして評価すると、最終的な処理結果が得られる。

例. リストとしての処理結果（先の例の続き）

```
>>> list( res )  Enter      ←リストとして処理結果を求める
[2, 4, 6]        ←処理結果
>>> list( res )  Enter      ←再度実行すると
[]               ←空になっている
```

この例からもわかるように、変数 `res` に得られたものは map オブジェクトであり、これはイテレータの一種であるため、参照した後は消滅する。

ちなみに、次のようにして一回の処理で結果を得ることもできる。

例. リストとしての処理結果 (2)（先の例の続き）

```
>>> list( map( dbl, [1,2,3] ) )  Enter      ←同様の処理を一度で行う
[2, 4, 6]        ←処理結果
```

2.10.1 map 関数

《 map 関数 》

書き方 1: `map(関数名, 対象のデータ構造)`

書き方 2: `map(関数名, 対象のデータ構造 1, 対象のデータ構造 2, ...)`

「対象のデータ構造」の全ての要素に対して「関数名」で表される関数の評価を実行するための **map オブジェクト**を返す。1 つの引数を取る関数の場合は「書き方 1」に、複数の引数を取る関数の場合は「書き方 2」に従うこと。

得られた map オブジェクトは `for` などの繰り返し処理のための**イテレータ**となる。また map オブジェクトを具体的なデータ構造に変換（`list` 関数でリストに変換するなど）することで実際の値が得られる。

リストの全要素に対して同じ関数を適用する処理を実現する場合、`for` 文を用いた繰り返し処理の形で記述することもできるが、map 関数を用いた方が記述が簡潔になるだけでなく、実行にかかる時間も概ね短くなる。

【サンプルプログラム】 map 関数と for 文の実行時間の比較

map 関数と for 文の実行時間を比較するためのサンプルプログラム map01.py を次に示す。これは、奇数／偶数を判定する関数 EvenOrOdd を、乱数を要素として持つリストに対して一斉適用する例である。

プログラム：map01.py

```
1  # coding: utf-8
2  # 必要なモジュールの読み込み
3  import time                # 時間計測用
4  from random import randrange # 乱数発生用
5
6  #####
7  # 偶数／奇数を判定する関数の定義
8  #####
9  def EvenOrOdd(n):
10     if n % 2 == 0:
11         return '偶'
12     else:
13         return '奇'
14
15  #####
16  # 試み(1)
17  #####
18  # 乱数リストの生成(1): 短いもの
19  lst = list( map( randrange, [100]*10 ) )
20
21  # 偶数／奇数の識別結果
22  lst2 = list(map(EvenOrOdd,lst))
23  print('10個の乱数の奇数／偶数の判定')
24  print(lst)
25  print(lst2,'\n')
26
27  #####
28  # 試み(2)
29  #####
30  # 乱数リストの生成(2): 1,000,000個
31  print('1,000,000個の乱数の奇数／偶数の判定')
32  t1 = time.time()
33  lst = list( map( randrange, [100]*1000000 ) )
34  t = time.time() - t1
35  print('乱数生成に要した時間:',t,'秒')
36
37  # 速度検査(1): forによる処理
38  lst2 = []
39  t1 = time.time()
40  for i in range(1000000):
41      lst2.append( EvenOrOdd(lst[i]) )
42  t = time.time() - t1
43  print('forによる処理:',t,'秒')
44
45  # 速度検査(2): mapによる処理
46  t1 = time.time()
47  lst2 = list(map( EvenOrOdd, lst )) # mapによる処理
48  t = time.time() - t1
49  print('mapによる処理:',t,'秒')
50
51  # 判定結果の確認
52  #print(lst2)
```

このプログラムを実行した例を次に示す。

10 個の乱数の奇数／偶数の判定

[92, 3, 7, 26, 11, 22, 76, 61, 79, 10]

['偶', '奇', '奇', '偶', '奇', '偶', '偶', '奇', '奇', '偶']

1,000,000 個の乱数の奇数／偶数の判定

乱数生成に要した時間: 0.9250545501708984 秒

for による処理: 0.32311391830444336 秒

map による処理: 0.15199923515319824 秒

map 関数による処理の方が for 文による処理よりも実行時間が小さいことがわかる。

2.10.1.1 複数の引数を取る関数の map

2つの引数の和を求める次のような関数 `wa` を考える。

例. 2つの引数の和を求める関数 `wa`

```
>>> def wa(a,b): return a+b  [Enter]    ←関数 wa の定義
...    [Enter]                ←定義の終了
>>> wa(2,3)  [Enter]          ←関数 wa の評価
5            ←結果
```

この関数の `map` による実行例を示す。

例. 関数 `wa` を `map` 関数で実行する（先の例の続き）

```
>>> list( map(wa,[1,2],[3,4]) )  [Enter]  ← map 関数に複数の引数（それぞれリスト）を与える
[4, 6]    ←結果
```

長さの異なるデータ列を `map` の引数に与えた場合は、長さが最も短いデータ列を基準に `map` の処理が行われる。（次の例参照）

例. 長さの異なるデータ列に対する `map`（先の例の続き）

```
>>> list( map( wa, [2], [3,4] ) )  [Enter]  ← map 関数に長さの異なるデータ列を与える
[5]    ←結果
```

短い方のリスト '`[2]`' の長さが基準になっていることがわかる。

2.10.1.2 map 関数に zip オブジェクトを与える方法

`map` 関数の処理対象のデータ列として **zip オブジェクト**¹⁶⁰ を与えることができる。以下に例を挙げて解説する。

例. 複数の引数を取る関数 `csv` を用いた `map`（その1）

```
>>> def csv( *arg ):  [Enter]    ←与えた引数（文字列）を','で連結する関数
...     return ','.join(arg)  [Enter]  ←連結処理
...    [Enter]    ←関数定義の記述の終了
>>> q1 = ['a','b','c','d']  [Enter]  ←1番目のデータ列
>>> q2 = ['1','2','3','4']  [Enter]  ←2番目のデータ列
>>> q3 = ['イ','ロ','ハ']  [Enter]  ←3番目のデータ列（短い）
>>> list( map(csv,q1,q2,q3) )  [Enter]  ←3つのデータ列に対する map
['a,1,イ','b,2,ロ','c,3,ハ']    ←処理結果
```

これと同等の処理を `zip` オブジェクトを用いて実現する。（次の例）

例. 複数の引数を取る関数 `csv` を用いた `map`（その2：先の続き）

```
>>> z = zip(q1,q2,q3)  [Enter]    ←zip オブジェクトの生成
>>> list( map(csv,*zip(*z)) )  [Enter]  ←zip オブジェクトに対する map 処理
['a,1,イ','b,2,ロ','c,3,ハ']    ←処理結果
```

`map` 関数にデータ列として `zip` オブジェクトを与えているが、その際 `*zip(*z)` としていることが重要である。これに関しては「2.6.1.10 `zip` 関数と `zip` オブジェクト」（p.94）でも取り上げているが、次の試みに関しても考察されたい。

例. 1つの試み（先の続き）

```
>>> z = zip(q1,q2,q3)  [Enter]    ←zip オブジェクトの生成
>>> list( map(csv,*z) )  [Enter]  ←zip オブジェクトに対する map 処理
['a,b,c','1,2,3','イ,ロ,ハ']    ←処理結果
```

`map` 関数にデータ列として `*z` を与えるとこのような処理結果となる。

¹⁶⁰ 「2.6.1.10 `zip` 関数と `zip` オブジェクト」（p.94）を参照のこと。

2.10.2 lambda と関数定義

引数として与えられた値（定義域）から別の値（値域）を算出して返すものを**関数**として扱うが、Python のプログラムの記述の中では**関数名のみ**を記述する場合もある。map 関数の中に記述する関数名もその 1 例であるが、この「関数名のみ」の表記は実体としては**関数オブジェクト**もしくは `lambda` である。

■ 実体としての関数

map 関数の解説のところで示したサンプルプログラムで扱った偶数／奇数を判定する関数 `EvenOrOdd` について考える。この関数の定義を次のようにして Python インタプリタに与える。

例. Python インタプリタに関数定義を与える

```
>>> def EvenOrOdd(n):  ←定義の記述の開始
...     if n % 2 == 0: 
...         return '偶' 
...     else: 
...         return '奇' 
...  ←定義の記述の終了（改行のみ）
>>>  ←Python コマンドラインに戻る
```

この関数の評価を実行する際、次のようにして引数を括弧付きで与える。

例. 上で定義した関数の評価（先の例の続き）

```
>>> EvenOrOdd(13)  ←評価の実行
'奇'  ←評価結果
```

次に、`'EvenOrOdd'` という記号そのものには何が割当てられているのかを確認する。

例. 関数名のみを参照する（先の例の続き）

```
>>> print( EvenOrOdd )  ←内容の確認
<function EvenOrOdd at 0x000001D863C63E18>  ←関数オブジェクト
```

このように、記号 `'EvenOrOdd'` には**関数オブジェクト**が割当てられていることがわかる。このオブジェクトは、データとして変数に割り当てることができるものであり、このような扱いができるオブジェクトは**第一級オブジェクト**¹⁶¹（first-class object）と呼ばれる。この例で扱っている関数オブジェクトも、他の変数に割り当てて使用すること（評価、実行）が可能である。（次の例を参照のこと）

例. 変数への関数オブジェクトの割当てと実行（先の例の続き）

```
>>> f = EvenOrOdd  ←別の変数 f に関数オブジェクトを割当てる
>>> f(13)  ←関数 f を実行
'奇'  ←値が得られている
```

■ lambda 式

関数の定義を `lambda`¹⁶² 式として記述して取り扱うことができる。先に例示した「2 倍の値を返す関数」`dbl` を `lambda` 式で実装する例を示す。

例. 値を 2 倍する関数 `dbl2` の実装例

```
>>> dbl2 = lambda n:2*n  ← lambda 式による実装
>>> dbl2(3)  ←実行
6  ←結果が得られている
```

この例の `dbl2` の内容を確認すると次のようになる。

¹⁶¹変数に値として割り当てることができ、プログラムの実行時にデータとして生成と処理ができるオブジェクトを指す。

関数定義を第一級オブジェクトとして扱える言語処理系は少なく、一部のリスト処理系（LISP の各種実装）がそれを可能にしている。

¹⁶²数学の関数表記 $f(x)$ の f を実体として扱い、対象となるデータにそれを適用して値を算出する**ラムダ計算**が由来である。

ラムダ計算に関しては提唱者 A. チャーチの著書 “The Calculi of Lambda Conversion”, *Princeton University Press, 1941* を参照のこと。

例. 上記作成の db12 を参照する（先の例の続き）

```
>>> db12 Enter          ← db12 の内容確認  
<function <lambda> at 0x00000210D6EB3E18> ←結果
```

このように関数オブジェクト（lambda 式）が格納されていることがわかる。

《 lambda 式の記述 》

書き方 1: lambda 仮引数: 戻り値の式

書き方 2: lambda 仮引数並び: 戻り値の式

書き方 3: lambda: 戻り値の式 (仮引数を取らないケース)

「仮引数並び」はコンマ ',' で区切る。

複数の仮引数を取る関数 wa の記述例を次に示す。

例. 2つの引数の和を求める関数 wa の実装

```
>>> wa = lambda a,b : a+b Enter      ← lambda 式を wa に割り当てる  
>>> wa(2,3) Enter      ←実行  
5          ←結果
```

また、次の例のように lambda 式を記号に割り当てずに適用することもできる。

例. lambda 式を直接引数に適用する。

```
>>> (lambda a,b : a+b)(2,3) Enter    ← lambda 式の直接適用  
5          ←結果
```

■ lambda と関数定義の違い: __name__ 属性

def 文による関数定義と lambda 式の記述は本質的に同じものであるように見えるが、それらの若干の相違点について考察する。そのために同様の機能を持つ関数を 2 種類用意する。

例. 同じ機能を def 文と lambda でそれぞれ実装する

```
>>> def f1(x,y): return x+y Enter    ← def 文による定義  
... Enter  
>>> f2 = lambda x,y: x+y Enter    ← lambda 式による定義
```

この例で定義された f1, f2 共に加算を実行するものである。

例. 上記 f1, f2 の実行（先の例の続き）

```
>>> f1(2,3) Enter    ← f1 の実行  
5          ←計算結果  
>>> f2(2,3) Enter    ← f1 の実行  
5          ←計算結果
```

f1, f2 共に同様の処理を行っていることがわかる。次に f1, f2 の __name__ 属性が異なることを示す。

例. __name__ 属性を調べる（先の例の続き）

```
>>> f1.__name__ Enter    ← f1 の __name__ 属性は  
'f1'          ←関数名 f1  
>>> f2.__name__ Enter    ← f2 の __name__ 属性は  
'<lambda>'
```

2.10.3 filter

データ列の要素の内、指定した条件を満たす要素のみを取り出す関数に filter がある。

《 filter 関数 》

書き方： filter(条件判定用の関数名, 対象のデータ構造)

「対象のデータ構造」の要素の内、「条件判定用の関数」の評価結果が真 (True) となるものを抽出して返す。戻り値は filter オブジェクトであり、これを具体的なデータ構造に変換 (list 関数でリストに変換するなど) することで実際のデータが得られる。

第一引数には関数名の他, lambda 式も指定できる。

整数を要素として持つリストの中から偶数の要素のみを取り出す処理を例として示す。

例. 偶数の取り出し

```
>>> from random import randrange  Enter  ←乱数発生用のライブラリの読み込み
>>> lst = list( map( randrange, [100]*17 ) )  Enter  ←乱数リスト (17 要素) の生成
>>> lst  Enter  ←内容確認
[15, 74, 64, 81, 58, 18, 70, 88, 45, 44, 3, 81, 36, 98, 57, 18, 27]  ←乱数リスト
>>> lst2 = list( filter( lambda n:n%2==0, lst ) )  Enter  ←偶数のみの抽出
>>> lst2  Enter  ←内容確認
[74, 64, 58, 18, 70, 88, 44, 36, 98, 18]  ←偶数のみのリスト
```

2.10.4 3 項演算子としての if~else...

条件分岐のための「if~else...」文に関しては「2.6.4 条件分岐」(p.97) のところで解説したが、3 項演算子としての「if~else...」の記述も可能である。

《 if~else... 演算子 》

書き方： 値 1 if 条件式 else 値 2

これは式であり、結果として値を返す。「条件式」が真の場合は「値 1」を、偽の場合は「値 2」を返す。

この式は、条件による値の評価の選択を lambda 式などの中で簡潔に記述するのに役立つ。

例. 偶数/奇数を判定する lambda 式

```
>>> evod = lambda n : '偶' if n % 2 == 0 else '奇'  Enter  ← 1 行で条件分岐を記述
>>> evod( 2 )  Enter  ← 2 は偶数か奇数か？
'偶'  ←処理結果
>>> evod( 3 )  Enter  ← 3 は偶数か奇数か？
'奇'  ←処理結果
```

2.10.5 all, any による一括判定

リストやタプルといったデータ構造に格納された真理値を一括で評価するには all 関数や any 関数を用いる。

《all 関数》

書き方： all(真理値を要素とするデータ構造)

「真理値を要素とするデータ構造」の全ての要素が True の場合に True を、それ以外の場合は False を返す。

例. all 関数

```
>>> p = [True, True, True, True]  Enter  ←全て True のリストに対する
>>> all( p )  Enter  ← all 関数の結果は
True  ← True (真) となる
>>> p = [True, True, False, True]  Enter  ← 1 つでも False があると
>>> all( p )  Enter  ← all 関数の結果は
False  ← False (偽) となる
```

《any 関数》

書き方： any(真理値を要素とするデータ構造)

「真理値を要素とするデータ構造」に1つでも True の要素が含まれる場合に True を、それ以外の場合は False を返す。

例. any 関数

```
>>> p = [False,False,True,False] Enter ← 1つでも True があると
>>> any( p ) Enter ← any 関数の結果は
True ← True (真) となる
>>> p = [False,False,False,False] Enter ← 全て False のリストに対する
>>> any( p ) Enter ← any 関数の結果は
False ← False (偽) となる
```

これらの関数が対象とするデータは真理値であるが、map 関数と組み合わせると、真理値以外の多量のデータに対して指定した条件を満たすかどうかを一括して検査することが可能となる。これに関してサンプルプログラム allany01.py を例に挙げて説明する。

プログラム：allany01.py

```
1 # coding: utf-8
2 # all, any のサンプル
3 import random
4 #-----#
5 # テストデータの作成 #
6 #-----#
7 # 全ての要素が50未満の乱数データのリスト
8 d1 = [ random.randrange(0,50) for x in range(7) ]
9 # 1つだけ50より大きい要素を持つリスト
10 d2 = [ random.randrange(0,50) for x in range(7) ]
11 d2[5] = 99
12
13 #-----#
14 # テスト関数 (50未満かどうかのチェック) #
15 #-----#
16 # テスト関数1 (50未満かどうかのチェック)
17 def chkU50( n ):
18     return n < 50
19
20 # テスト関数2 (50以上かどうかのチェック)
21 def chkG50( n ):
22     return n >= 50
23
24 # 汎用テスト関数
25 def forAll( f, d ): # dの全てがfか?
26     return all( map(f,d) )
27
28 def ifAny( f, d ): # dにfなるものがあるか?
29     return any( map(f,d) )
30
31 #-----#
32 # 検査の実行 #
33 #-----#
34 print( 'データ1:', d1 )
35 print( '全て50未満か?:', forAll( chkU50, d1 ) )
36 print( '50以上があるか?:', ifAny( chkG50, d1 ), '\n' )
37 print( 'データ2:', d2 )
38 print( '全て50未満か?:', forAll( chkU50, d2 ) )
39 print( '50以上があるか?:', ifAny( chkG50, d2 ) )
```

このプログラムは、乱数のリストに対して条件判定を一括して行うサンプルである。8行目では「50未満の整数」の乱数を要素として持つデータ d1 を、10～11行目では1つだけ50を超える整数を持つ乱数データ d2 を生成している。

17～18行目では「50未満」を判定する関数 chkU50 を、21～22行目では「50以上」を判定する関数 chkG50 を定義している。

25～26 行目では、指定した条件判定関数が与えられた「全てのデータで真となる」ことを判定する関数 `forAll` を、28～29 行目では、指定した条件判定関数が与えられたデータに対して「少なくとも 1 つ真となる」ことを判定する関数 `ifAny` を定義している。

`allany01.py` を実行した例を次に示す。

```
データ 1: [24, 32, 14, 26, 2, 33, 15]
全て 50 未満か?: True
50 以上があるか?: False
データ 2: [28, 40, 15, 25, 14, 99, 4]
全て 50 未満か?: False
50 以上があるか?: True
```

このサンプルで定義している `forAll`, `ifAny` 関数のような実装により、多量のデータに対する判定を行うための汎用の関数が定義できる。

2.10.5.1 使用上の注意：ネストされたデータ構造

`all` や `any` の引数に与えるデータ構造がネストされている場合は特に注意すること。(次の例)

例. `all` でネストされたデータ構造を判定する

```
>>> p = [True, True, [False], True]  Enter ← このようにネストされたリストを
>>> all( p )  Enter ← all で判定すると
True          ← 真となる
```

この例では、`p` の要素の 1 つが `False` になっているように見えるが、`[False]` となっていることにより、その要素自体が `bool` 型に解釈された場合に `True` となる。従って、`p` の全要素が `True` であると見做される。`any` についても同様に注意を要する。(次の例)

例. `any` でネストされたデータ構造を判定する

```
>>> p = [False, False, [False], False]  Enter ← このようにネストされたリストを
>>> any( p )  Enter ← any で判定すると
True          ← 真となる
```

この例では `p` の全ての要素が `False` であるように見えるが、要素の 1 つが `[False]` であり、それが `True` と見做される。

2.10.6 高階関数モジュール：functools

本書では `map`, `filter` といった関数を取り上げているが、`functools` モジュールは「関数をオブジェクトに対して実行する際の高度な機能」を提供している。このモジュールは Python 処理系に標準的に添付されており、

```
import functools
```

として読み込んで使用する。ここでは `functools` の使用例を 1 つ紹介する。

2.10.6.1 reduce

`reduce` は、2 つの引数をとる関数（あるいは `lambda`）をデータ列の要素に順番に累積的に適用する。(次の例参照)

例. 2 つの数の和を求める関数を用いて、データ列の合計を求める

```
>>> import functools  Enter ← モジュールの読み込み
>>> def wa(a,b): return a+b  Enter ← 2 つの引数の和を求める関数の定義
...  Enter ← 定義の記述の終了
>>> d = [1,2,3,4,5,6,7,8,9,10]  Enter ← 1～10 のデータを用意
>>> functools.reduce( wa, d )  Enter ← データに対して reduce で処理を実行
55          ← 処理結果
```

この例で定義されている関数 `wa(a,b)` は、`a+b` を求めるものであるが、`reduce` を使用すると `[a,b,c,d,e,f,g]` というデータ列に対して、

```
(((((a+b)+c)+d)+e)+f)+g
```

という処理を施す。

2.11 代入式

イコール「=」による変数への値の割り当ては文であり、それ自体は値を返さない。Python 3.8 からは、「:=」¹⁶³ を用いて変数に値を割り当てる**代入式**¹⁶⁴ が導入された。代入式は代入した値を返す。

例. 「=」と「:=」の違い

```
>>> x = (a=3) [Enter]    ←通常の代入文「a=3」を x に代入しようとする…
File "<stdin>", line 1    ←文法エラー (SyntaxError) となる
  x = (a=3)
  ~~~
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?

>>> x = (a:=3) [Enter]    ←代入式「a:=3」は x に代入することができる
>>> x [Enter]             ←内容確認
3                          ←結果表示
```

代入式を用いると、条件式の記述の中に代入処理を直接記述するといったことが可能となり、プログラムの可読性を高める場合がある。(次の例参照)

例. 標準入力（キーボード）から数値を入力するサイクル（0 で終了）

```
>>> while (v:=int(input('0 で終了>'))): [Enter]    ←条件判定の中に代入式を用いる
...     print(' 入力値:',v) [Enter]
... [Enter]    ←繰り返し文の終了
0 で終了>1 [Enter]    ←値の入力 (0 でない)
入力値: 1    ←出力
0 で終了>2 [Enter]    ←値の入力 (0 でない)
入力値: 2    ←出力
0 で終了>0 [Enter]    ←値の入力 (0 で終了)
>>>    ←対話モードのプロンプトに戻った
```

代入式を使わずにこれと同じ処理を記述すると次のようになる。

例. 上と同等の処理

```
>>> while True: [Enter]    ←繰り返しの記述の開始
...     v=int(input('0 で終了>')) [Enter]    ←通常の代入文で値を読み取る
...     if v != 0: [Enter]    ←値の判定
...         print(' 入力値:',v) [Enter]
...     else: [Enter]
...         break [Enter]
... [Enter]    ←繰り返し文の終了
0 で終了>1 [Enter]    ←値の入力 (0 でない)
入力値: 1    ←出力
0 で終了>2 [Enter]    ←値の入力 (0 でない)
入力値: 2    ←出力
0 で終了>0 [Enter]    ←値の入力 (0 で終了)
>>>    ←対話モードのプロンプトに戻った
```

¹⁶³ウォルラス演算子と呼ばれることもある。

¹⁶⁴PEP572 – Assignment Expressions

2.12 構造的パターンマッチング

Python 3.10 の版から高度な分岐処理である**構造的パターンマッチング**の構文（match～case 構文）¹⁶⁵ が導入された。本書では、これに関して基本的な事柄について解説する。

《構造的パターンマッチング》

書き方： match 値:

 case パターンの記述 1:

 (対象の処理 1)

 case パターンの記述 2:

 (対象の処理 2)

 :

 検査対象の「値」を「パターンの記述」に順番に照らし合わせ、マッチングが成功した部分の「対象の処理」（スイートの形式）を実行する。（C 言語の switch～case 構文のようなフォールスルーは起こらない）

これは一見すると C 言語や Java 言語の switch～case の構文に似ているが、それ以上の機能を実現する。この構文の最も単純な使用例をサンプルプログラム stpmatch01.py に示す。

プログラム：stpmatch01.py

```
1 # coding: utf-8
2 v = 1          # これを様々な値に変更して試す
3
4 match v:
5     case 1:
6         print('One')
7     case 2:
8         print('Two')
9     case 3:
10        print('Three')
11    case _:
12        print('Other')
```

これは、変数 v が保持する値を match～case 構文に与え、パターンマッチが成功する case 部のスイートを実行するものである。この例にあるように変数 v の値が 1 である場合、実行結果として「One」と出力される。（v の値を様々な値に変更して実行を試されたい）

このプログラムの 11 行目の「case _:」という記述は、それより上に記述されているパターンにマッチしなかった場合を意味する。この「_」（アンダースコア）の代わりに変数を記述して変数 v の値を受け取ることもできる。このことに関するサンプルプログラム stpmatch02.py を示す。

プログラム：stpmatch02.py

```
1 # coding: utf-8
2 v = 4          # これを様々な値に変更して試す
3
4 match v:
5     case 1:
6         print('One')
7     case 2:
8         print('Two')
9     case 3:
10        print('Three')
11    case x:
12        print('Other:', x)
```

このプログラムの 11 行目で「case x:」としているが、これにより、上の 3 つの case にマッチしなかった場合に、変数 v の値を変数 x に受け取ることができる。

この例の通り v の値が 4 である場合の実行結果の出力は「Other: 4」となる。

¹⁶⁵PEP622 – Structural Pattern Matching

2.12.1 構造的なパターン

case の後に記述するパターンとして、変数を含むデータ構造（**構造的なパターン**）を記述することができ、match に与えたデータ構造に対応する位置の値をパターン内の変数に受け取ることができる。このことをサンプルプログラム stpmatch03.py で示す。

プログラム：stpmatch03.py

```
1 # coding: utf-8
2 v = [1,2]          # リストの要素の個数を変更して試す
3
4 match v:
5     case [x]:
6         print('1要素のリスト： 要素 =', x )
7     case [x,y]:
8         print('2要素のリスト： 要素 =', x, y )
9     case [x,y,z]:
10        print('3要素のリスト： 要素 =', x, y, z )
11    case A:
12        print('それ以外：', A )
```

このプログラムでは、変数 v にリスト [1,2] を与えており、これが7行目の「case [x,y]」にマッチし、x に 1 が、y に 2 が得られる。従って、このプログラムを実行すると「2要素のリスト： 要素 = 1 2」が出力される。2行目を変更して変数 v に与えるリストの長さを様々に変えて実行を試みられたい。

構造的なパターンには具体的な値（**リテラル**）を含むことができる。このことに関してサンプルプログラム stpmatch03-2.py を示す。

プログラム：stpmatch03-2.py

```
1 # coding: utf-8
2 L = ['加算', 1, 2]
3 #L = ['乗算', 3, 4]
4 #L = ['総和', 1,2,3,4]
5 #L = ['総乗', 1,2,3,4]
6 #L = ['関数名', 5,6,7,8]
7
8 match L:
9     case ['加算',x,y]:
10        print(L,'->', x+y )
11     case ['乗算',x,y]:
12        print(L,'->', x*y )
13     case ['総和', *a]:
14        r = 0
15        for x in a: r = r + x
16        print(L,'->', r )
17     case ['総乗', *a]:
18        r = 1
19        for x in a: r = r * x
20        print(L,'->', r )
21     case A:
22        print(L,'->', A )
```

このプログラムでは変数 L にリストを与えているが、その第1要素は計算処理を意味する文字列とし、残りの要素の値を用いて計算処理を実行するもの¹⁶⁶とする。この例では変数 L に ['加算', 1, 2] を与えており「1と2を加算する」処理を意図している。このプログラムを実行すると L の値が9行目の case にマッチし、「['加算', 1, 2] -> 3」が出力される。

このプログラムの2～6行目の部分のコメント「#」のどれか1つを外すことで実行結果が様々に変わることを確認されたい。

参考 match の次に与える値と case の次に記述するパターンが異なるデータ構造（例えばリストとタプルなど）であってもマッチすることがある。

¹⁶⁶代表的な Lisp 言語における S 式による表現に倣った。

2.12.2 条件付きのパターンマッチング

case の後のパターンに if の記述を付けることで条件（guard）を設定することができる。このことに関するサンプルプログラム stpmatch04.py を示す。

プログラム：stpmatch04.py

```
1 # coding: utf-8
2 v = 3          # これを様々な値に変更して試す
3
4 match v:
5     case p if p > 0:
6         print(p, 'は正の値です。')
7     case p if p < 0:
8         print(p, 'は負の値です。')
9     case p:
10        print(p, 'はゼロです。')
```

このプログラムでは変数 `v` に 3 という値を与えているが、これが 5 行目の case にマッチする。この行のパターンの後ろには「if `p > 0`」が記述されており、この条件を満たすので、このプログラムを実行すると「3 は正の値です。」と出力される。

このプログラムの 2 行目を変更して変数 `v` に与える値を様々に変えて実行を試みられたい。

2.12.3 クラスのインスタンスに対するパターンマッチング

あるクラスのインスタンスに対するパターンマッチングが可能であり、インスタンス変数（属性）の値を判りやすい形で取り出すことができる。このことに関するサンプルプログラム stpmatch05.py を示す。

プログラム：stpmatch05.py

```
1 # coding: utf-8
2 import math
3
4 class Pos:      # xy座標上の点を表すクラス
5     def __init__(self, ix, iy):
6         self.x = ix
7         self.y = iy
8     def show(self):
9         print('( ', self.x, ', ', self.y, ' )')
10    def norm(self):
11        print('原点からの距離:', math.sqrt(self.x**2+self.y**2))
12
13 p = Pos(3, -4)      # x成分, y成分の値を様々に変更して試す
14 p.show(); p.norm()
15
16 match p:
17     case Pos(x=0, y=0):
18         print('原点です。')
19     case Pos(x=v1, y=_) if v1==0:
20         print('y軸上にあります。')
21     case Pos(x=_, y=v2) if v2==0:
22         print('x軸上にあります。')
23     case Pos(x=v1, y=v2) if v1>0 and v2>0:
24         print('第一象限にあります。')
25     case Pos(x=v1, y=v2) if v1<0 and v2>0:
26         print('第二象限にあります。')
27     case Pos(x=v1, y=v2) if v1<0 and v2<0:
28         print('第三象限にあります。')
29     case Pos(x=v1, y=v2) if v1>0 and v2<0:
30         print('第四象限にあります。')
```

このプログラムで定義されているクラス `Pos` は x - y 座標上の位置を表す（図 9）ものであり、このクラスのインスタンスに対するパターンマッチングを行っている。

このプログラムに記述されているように、あるクラスのインスタンスの状態（属性の値）をパターンとして表現するには case の記述を次のようにする。

case クラス名 (属性名 1=変数 1, 属性名 2=変数 2, …): (if による guard の記述も可能)

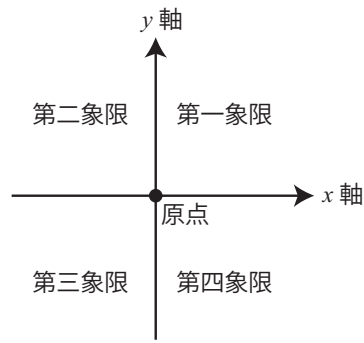


図 9: x - y 座標上の領域

この記述によって、match に与えられたインスタンスの「属性名」で示される属性の値を「変数」に受け取ることができる。またこの記述の際「変数」の代わりに具体的な値を記述して、それをパターンとすることもできる。(17 行目の記述)

このプログラムでは第四象限の点 $(3, -4)$ が Pos クラスのインスタンス p として作成されており、それに対してパターンマッチングを行っている。このプログラムを実行すると

```
( 3 , -4 )  
原点からの距離:  5.0  
第四象限にあります。
```

と出力される。

プログラムの 13 行目を変更して Pos クラスのコンストラクタ に与える値を様々に変えて実行を試みられたい。

3 KivyによるGUIアプリケーションの構築

Kivy は MIT ライセンスで配布される GUI ライブラリであり、インターネットサイト <https://kivy.org/> から入手できる。インストール方法から API の説明まで当該サイトで情報を入手することができる。本書では Kivy の基本的な使用方法について解説する¹⁶⁷。

3.1 Kivy の基本

Kivy によるアプリケーションプログラムは、App クラスのオブジェクトとして構築する。App クラスの使用に際して、下記のようにして必要なモジュールを読み込んでおく。

```
from kivy.app import App
```

3.1.1 アプリケーションプログラムの実装

具体的には、App クラスかそれを継承する（拡張する）クラス（以後「アプリケーションのクラス」と呼ぶ）をプログラマが定義し、そのクラスのインスタンスを生成することでアプリケーションプログラムが実装できる。アプリケーションのインスタンスに対して run メソッドを実行することでアプリケーションプログラムの動作が開始する。

run メソッドを呼び出すと、最初にアプリケーションのクラスのメソッド build が呼び出される。この build メソッドは、App クラスに定義されたメソッドであり、これをプログラマが上書き定義（オーバーライド）する形で、アプリケーション起動時の処理を記述する。build メソッドで行うことは主に GUI の構築などである。

Kivy によるアプリケーションプログラム構築の素朴な例として、サンプルプログラム kivy01-1.py を示す。

プログラム：kivy01-1.py

```
1 # coding: utf-8
2 from kivy.app import App          # 基本となるアプリケーションクラス
3 from kivy.ui.label import Label   # ラベルオブジェクト
4
5 # アプリケーションのクラス
6 class kivy01(App):
7     def build(self):
8         self.lb1 = Label(text='This is a test of Kivy.')
9         return self.lb1
10
11 #---- メインルーチン ----
12 # アプリケーションのインスタンスを生成して起動
13 ap = kivy01()
14 ap.run()
```

この例では、App クラスを拡張した kivy01 クラスとしてアプリケーションを構築している。kivy01 クラスの中では build メソッドをオーバーライド定義しており、Label ウィジェット（文字などを表示するウィジェット¹⁶⁸）を生成している。

このプログラムを実行すると図 10 のようなウィンドウが表示される。

■ Kivy 利用時のトラブルに関すること

Kivy を使用したプログラムを実行する際に、システムが使用するグラフィックス用 API の関係でエラーが発生する場合がある。これに関しては巻末付録「B.1 Kivy 利用時のトラブルを回避するための情報」（p.406）を参照されたい。

3.1.2 GUI 構築の考え方

先のプログラム kivy01-1.py においては、GUI 要素として Label ウィジェットを生成しており、このウィジェットを build メソッドの戻り値としている。Kivy ではこのように、GUI の最上位のウィジェットを build メソッドの戻り値とする。

¹⁶⁷巻末付録「B Kivy に関する情報」（p.406）も参照のこと

¹⁶⁸ウィジェット：GUI を構成する部品のこと。

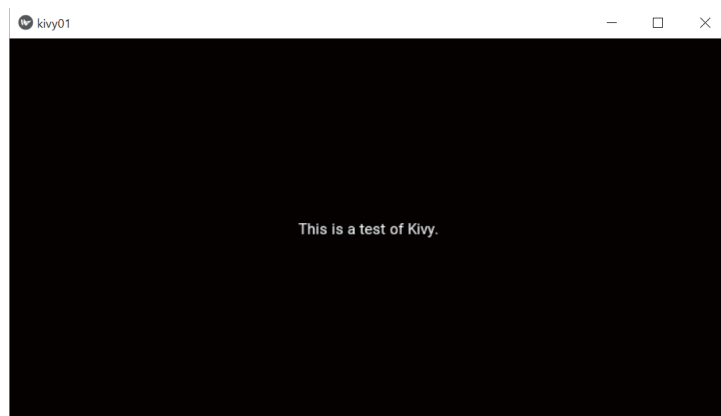


図 10: Kivy01-1.py の実行結果

Kivy では、GUI を階層構造として構築する。すなわち「親」のオブジェクトの配下に「子」のオブジェクト群が従属する形で GUI を構築する。Kivy には GUI の配置を制御する Layout や Screen といったクラスがあり、それらクラスのオブジェクト配下にウィジェットなどの要素を配置する形式で GUI を構築する。先のプログラム kivy01-1.py では、最も単純に GUI の導入説明をするために、Label オブジェクトが 1 つだけ存在するものとした。すなわち、この Label オブジェクトが GUI の最上位の「親オブジェクト」となっている。実際のアプリケーション構築においては、Layout や Screen のオブジェクトなど、要素の配置のためのものを GUI の最上位オブジェクトとすることが一般的である。

Kivy における GUI のためのクラスは、大まかにウィジェット、レイアウト、スクリーンの 3 つに分けて考えることができる。

3.1.2.1 Widget (ウィジェット)

ボタンやラベル、チェックボックス、テキスト入力エリアといった基本的な要素である。代表的なウィジェットを表 28 に挙げる。

表 28: 代表的なウィジェット

クラス	機能
Label	ラベル（文字などを表示する）
Button	ボタン
TextInput	テキスト入力（フィールド／エリア）
CheckBox	チェックボックス
ProgressBar	進捗バー
Slider	スライダ
Switch	スイッチ
ToggleButton	トグルボタン
Image	画像表示
Video	動画表示

この他にも多くのウィジェットがあるが、詳しくは Kivy の公式サイトを参照のこと。

3.1.2.2 Layout (レイアウト)

レイアウトは GUI オブジェクトを配置するためのもので、一種の「コンテナ」（容器）と考えることができる。例えば BoxLayout を使用すると、その配下にウィジェットなどを水平あるいは垂直に配置する（図 11）ことができる。

例えば、BoxLayout を入れ子の形で（階層的に）組み合わせると、図 12 のような GUI デザインを実現することができる。

利用できるレイアウトを表 29 に挙げる。

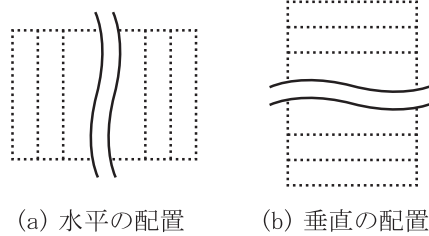


図 11: BoxLayout

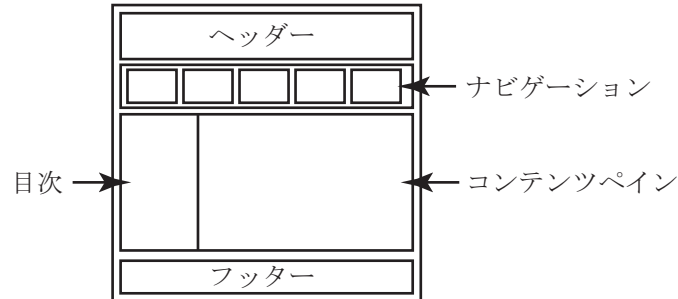


図 12: BoxLayout を組み合わせたデザインの例

表 29: 利用できるレイアウト

クラス	機能
BoxLayout	水平／垂直のレイアウト
GridLayout	縦横（2次元）のグリッド配置
StackLayout	水平あるいは垂直方向に順番に追加される配置
AnchorLayout	片寄せ，中揃え（均等配置）の固定位置
FloatLayout	直接位置指定（絶対，相対）
RelativeLayout	直接位置指定（画面の位置：絶対，相対）
PageLayout	複数ページの切り替え形式
ScatterLayout	移動，回転などを施すためのレイアウト

■ BoxLayout によるレイアウトの例

サンプルプログラム kivy01-BoxLayout01.py によって BoxLayout による GUI のレイアウトを例示する。

プログラム：kivy01-BoxLayout01.py

```

1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  import sys
4  from kivy.app import App
5  from kivy.uix.boxlayout import BoxLayout
6  from kivy.uix.button import Button
7  from kivy.core.window import Window
8
9  #----- ウィンドウの設定 -----
10 Window.size = (300,300)
11
12 #----- 最上位のレイアウト -----
13 root = BoxLayout(orientation='vertical')
14
15 #----- GUIの構築 -----
16 row1 = BoxLayout(orientation='horizontal') # 1行目のレイアウト
17 row1.size_hint = (1.0,0.25) # 比率による高さの設定
18 b11 = Button(text='11'); b11.size_hint = ( 0.2, 1.0 ) # 比率による
19 b12 = Button(text='12'); b12.size_hint = ( 0.8, 1.0 ) # 横幅の設定
20 row1.add_widget(b11); row1.add_widget(b12)
21 root.add_widget(row1)
22
23 row2 = BoxLayout(orientation='horizontal') # 2行目のレイアウト
24 row2.size_hint = (1.0,0.75) # 比率による高さの設定

```

```

25 b21 = Button(text='21');      b21.size_hint = ( 2, 1.0 ) # 整数比
26 b22 = Button(text='22');      b22.size_hint = ( 3, 1.0 ) # による
27 b23 = Button(text='23');      b23.size_hint = ( 5, 1.0 ) # 横幅の設定
28 row2.add_widget(b21);      row2.add_widget(b22);      row2.add_widget(b23)
29 root.add_widget(row2)
30
31 #----- アプリケーションのクラス -----
32 class kivy01(App):
33     def build(self):
34         return root
35
36 #----- アプリケーションの起動 -----
37 ap = kivy01()
38 ap.run()

```

このプログラムは BoxLayout を入れ子にしてボタン¹⁶⁹ を縦横に配置するものである。このプログラムを実行すると図 13 の (a) のようなウィンドウが表示される。

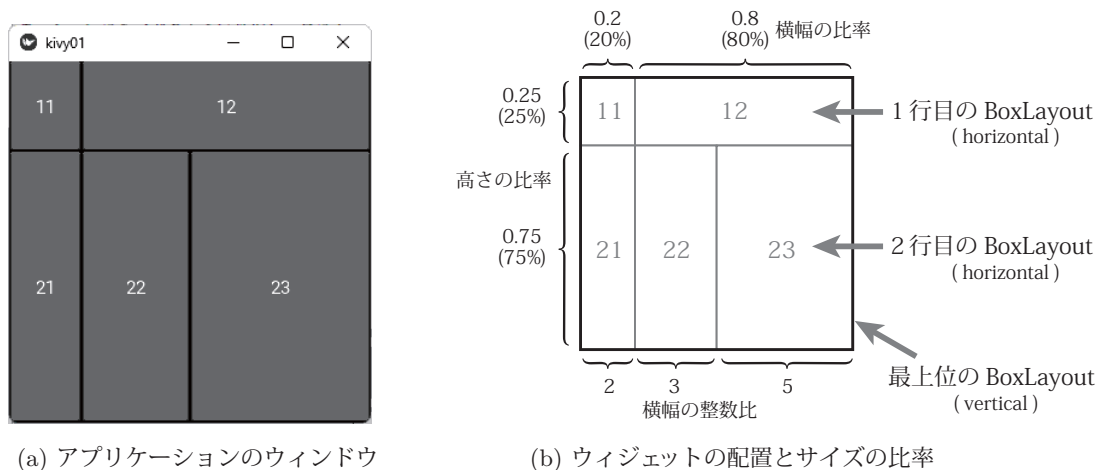


図 13: kivy01-BoxLayout01.py を実行した様子

このプログラムは垂直配置の BoxLayout である root (13 行目で作成) を最上位のウィジェットとして、その中に水平配置の BoxLayout である row1, row2 (16,23 行目で作成) を子ウィジェットとして保持 (21,29 行目で登録) している。row1 の中にはボタン b11,b12 が、row2 の中にはボタン b21,b22,b23 が水平方向に配置されている。

ウィジェットのサイズ指定の方法：

各ウィジェットのサイズは size_hint プロパティで制御する。size_hint プロパティには (横比率, 縦比率) の形式で親ウィジェットに対する比率として値を与える。例えば 2 つの BoxLayout である row1, row2 の高さの比は 17 行目と 24 行目にあるように 0.25:0.75 として値を設定している。同様に 2 つのボタン b11, b12 の横幅の比率は 18 行目と 19 行目にあるように 0.2:0.8 として設定している。また size_hint プロパティの値はより柔軟に解釈され、ボタン b21, b22, b23 の size_hint の様な形式 (合計が 1.0 にならないような形) で与えること (25~27 行目) もできる。

ウィジェットの size_hint の設定を行わない場合は、システムがサイズを自動的に調整する。上のプログラムで size_hint の設定を全て抹消すると、全てのウィジェットが均等なサイズでレイアウトされるので試されたい。

3.1.2.3 Screen (スクリーン)

スクリーンにはレイアウトやウィジェットを配置することができ、1 つのスクリーン (Screen) は 1 つの操作パネルと見ることができる。更に複数のスクリーンをスクリーンマネージャ (ScreenManager) と呼ばれるオブジェクト配下に設置することができ、それらスクリーンを切り替えて表示することができる。

スクリーンマネージャを用いて構築された GUI は、いわゆるプレゼンテーションスライドのように動作し、各スクリーンを遷移して切り替えること (transition) で異なる複数のインターフェースを切り替えることができる。

Screen, ScreenManager の扱いに関しては「3.9 GUI 構築の形式」(p.224) のところで説明する。

¹⁶⁹ 「3.3.2 ボタン：Button」(p.210) で解説する。

3.1.3 ウィンドウの扱い

ウィンドウ (Window) は構築する GUI アプリケーション全体を表すオブジェクトであり、1つのアプリケーションに1つのウィンドウオブジェクトが存在する。

ウィンドウオブジェクトを明に扱うには、次のようにして Window モジュールを読み込んでおく必要がある。

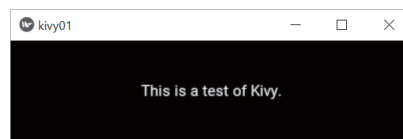
```
from kivy.core.window import Window
```

Window オブジェクトに対して、ウィンドウサイズの指定をはじめとする各種制御ができる。先のプログラムを改変した kivy01-2.py を次に示す。

プログラム：kivy01-2.py

```
1  # coding: utf-8
2  from kivy.app import App                # 基本となるアプリケーションクラス
3  from kivy.ui.label import Label         # ラベルオブジェクト
4  from kivy.core.window import Window     # ウィンドウの操作に必要なもの
5
6  # アプリケーションのクラス
7  class kivy01(App):
8      def build(self):
9          self.lb1 = Label(text='This is a test of Kivy.')
10         return self.lb1
11
12  #---- メインルーチン ----
13  # ウィンドウサイズの設定 (400×100)
14  Window.size = (400,100)
15  # アプリケーションのインスタンスを生成して起動
16  ap = kivy01()
17  ap.run()
```

このように Window オブジェクトの size プロパティに値を設定することで、アプリケーションのウィンドウサイズを変更することができる。(図 14 参照)



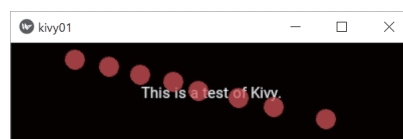
Window の size プロパティを 400 × 100 に設定している

図 14: Kivy01-2.py の実行結果

Window のプロパティとしてはこの他にも clearcolor もあり、この値¹⁷⁰を設定することでウィンドウの色を変更することもできる。

3.1.4 マルチタッチの無効化

Kivy ではポインティングデバイスのマルチタッチを受け付けるが、通常の PC のデスクトップ環境ではマルチタッチが使用できないことが多い。この場合、Kivy アプリケーションのウィンドウ内でマウスを右クリックすると、ウィンドウ内に小さな円が表示される。先に挙げたプログラム kivy01-2.py において実行中にマウスを右クリックした様子の例を図 15 に示す。



右クリックした場所に小さな円が表示される

図 15: 通常の PC 環境での右クリック

¹⁷⁰clearcolor の値は (R,G,B, α) のタプルで与える。この場合の各要素は 0～1 の数値である。

ポインティングデバイスのマルチタッチを無効化するとこの反応はなくなる。そのためには、Kivy 関連モジュールの読み込みに先立って次のように記述する。

```
from kivy.config import Config
Config.set('input', 'mouse', 'mouse,disable_multitouch')
```

Kivy には各種の設定のための Config オブジェクトがあり、それに対して set メソッドを用いることで各種の設定作業を行う。

3.2 基本的な GUI アプリケーション構築の方法

3.2.1 イベント処理（導入編）

GUI アプリケーションプログラムはユーザからの操作をはじめとするイベントの発生を受けてイベントハンドラを起動する、いわゆるイベント駆動型のスタイルを基本とする。ここではサンプルプログラムの構築を通してイベント処理の基本的な実装方法について説明する。

先に挙げたサンプルプログラムを更に改変して、ラベルオブジェクトにタッチ¹⁷¹ が起こった際のイベント処理について説明する。

3.2.1.1 イベントハンドリング

ここでは、Label オブジェクトが touch_down（タッチの開始／ボタンの押下）、touch_move（ドラッグ）、touch_up（タッチの終了／ボタンを放す）といったイベントを受け付ける例を挙げて説明する。ウィジェットには、それらイベントを受け付けるためのメソッド（on_touch_down, on_touch_move, on_touch_up）が定義されており、プログラムはウィジェットの拡張クラスを定義して、それらメソッドをオーバーライドして、実際の処理を記述する。

プログラム：kivy01-3.py

```
1  # coding: utf-8
2  from kivy.app import App                # 基本となるアプリケーションクラス
3  from kivy.uix.label import Label        # ラベルオブジェクト
4  from kivy.core.window import Window    # ウィンドウの操作に必要なもの
5
6  # ラベルの拡張
7  class MyLabel(Label):
8      # タッチ開始（マウスボタンの押下）の場合の処理
9      def on_touch_down(self, touch):
10         print(self.events())
11         print('touch down: ', touch.spos)
12     # タッチの移動（ドラッグ）の場合の処理
13     def on_touch_move(self, touch):
14         print('touch move: ', touch.spos)
15     # タッチ終了（マウスボタンの解放）の場合の処理
16     def on_touch_up(self, touch):
17         print('touch up : ', touch.spos)
18
19 # アプリケーションのクラス
20 class kivy01(App):
21     def build(self):
22         self.lb1 = MyLabel(text='This is a test of Kivy.')
23         return self.lb1
24
25 #---- メインルーチン ----
26 # ウィンドウサイズの設定
27 Window.size = (400,100)
28 # アプリケーションのインスタンスを生成して起動
29 ap = kivy01()
30 ap.run()
```

解説：

このプログラムでは、Label クラスを拡張した MyLabel クラスを定義し（7～17 行目）、そのクラスでイベントハ

¹⁷¹ パーソナルコンピュータの操作環境ではマウスのクリックがこれに相当する。Kivy はスマートフォンやタブレットコンピュータでの処理を前提としており、タッチデバイスを基本にしたイベント処理となっている。

ンドリングをしている。このクラスではそれぞれのイベントに対応するメソッドの記述をしており、仮引数として記述された `touch` に、イベント発生時の各種情報を保持するオブジェクトが与えられる。

アプリケーション全体は、App クラスの拡張クラス `kivy01` クラスとして定義されている。(20～23 行目)。10 行目にあるように、`events` メソッドを実行すると、そのオブジェクトが対象とするイベントの一覧情報が得られる。また、マウスやタッチデバイスのイベントが持つ `spox` プロパティには、イベントが発生した位置の情報が保持されている。

このプログラムを実行した際の標準出力の例を次に示す。

```
dict_keys([on_ref_press, on_touch_down, on_touch_move, on_touch_up])
touch down: (0.0875, 0.45999999999999996)
touch move: (0.0875, 0.48)
touch move: (0.09, 0.48)
touch move: (0.0925, 0.48)
touch move: (0.095, 0.48)
touch move: (0.0975, 0.48)
touch move: (0.1, 0.48)
touch move: (0.0975, 0.47)
touch up : (0.0975, 0.47)
```

3.2.2 アプリケーション構築の例

ここでは、簡単な事例を挙げて、GUI アプリケーションプログラムの構築について説明する。事例として示すアプリケーションは簡単な描画アプリケーションで、概観を図 16 に示す。このアプリケーションは、ウィンドウ内の描画領域にタッチデバイスやマウスでドラッグした軌跡を描画する。またウィンドウ上部の「Clear」ボタンをタッチ（クリック）すると描画領域を消去し、「Quit」ボタンをタッチ（クリック）するとアプリケーションが終了する。

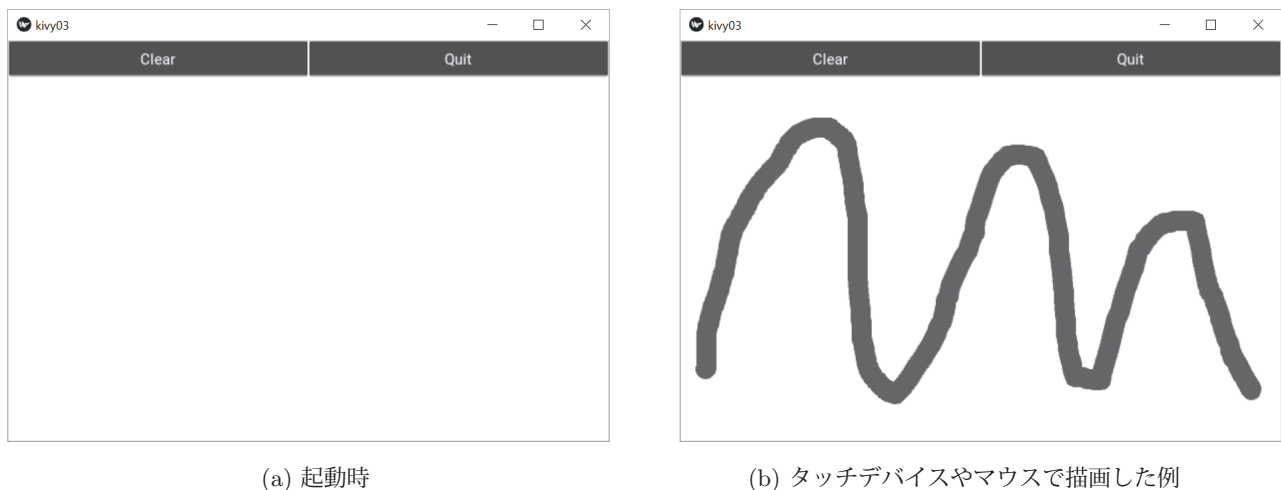


図 16: kivy03.py の実行結果

この事例では、`BoxLayout` を使用してウィンドウ内のレイアウトを実現している。具体的には、水平方向の `BoxLayout` を用いて「Clear」と「Quit」の2つのボタンを配置し、そのレイアウトと描画領域を垂直方向の `BoxLayout` で配置している。`BoxLayout` 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んでおく。

BoxLayout を使用するためのモジュールの読み込み

```
from kivy.uix.boxlayout import BoxLayout
```

`BoxLayout` の配置方向（水平、垂直）の指定は、インスタンス生成時にコンストラクタの引数としてキーワード引数「`orientation=`」を与える。この引数の値として「`horizontal`」を与えると水平方向、「`vertical`」を与えると垂直方向の配置となる。

ボタンウィジェット（`Button`）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んでおく。

Button を使用するためのモジュールの読み込み

```
from kivy.uix.button import Button
```

ボタンウィジェットのトップに表示する文字列は、インスタンス生成時にコンストラクタの引数としてキーワード引数「`text=`」を与える。この引数の値として文字列を与えると、それがボタントップに表示される。

グラフィックスを描画するには、Widget クラスを使用するのが一般的である。このクラスは多くのウィジェットの上位クラスであり、描画をするための canvas という要素を持つ。グラフィックスの描画はこの canvas に対して行う。Widget 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んでおく。

Widget を使用するためのモジュールの読み込み

```
from kivy.uix.widget import Widget
```

【実装例】

今回の事例のプログラムを kivy03.py に挙げる。これを示しながら GUI アプリケーションプログラム構築の流れを説明する。

プログラム：kivy03.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  import sys
4  from kivy.app import App
5  from kivy.uix.boxlayout import BoxLayout
6  from kivy.uix.button import Button
7  from kivy.uix.widget import Widget
8  from kivy.graphics import Color, Line
9  from kivy.core.window import Window
10
11 #----- 拡張クラスの定義 -----
12 # アプリケーションのクラス
13 class kivy03(App):
14     def build(self):
15         return root
16
17 # Clear ボタンのクラス
18 class BtnClear(Button):
19     def on_release(self):
20         drawArea.canvas.clear()
21 # Quit ボタンのクラス
22 class BtnQuit(Button):
23     def on_release(self):
24         sys.exit()
25
26 # 描画領域のクラス
27 class DrawArea(Widget):
28     def on_touch_down(self,t):
29         self.canvas.add( Color(0.4,0.4,0.4,1) ) # 描画色の設定
30         self.lineObject = Line( points=(t.x,t.y), width=10 )
31         self.canvas.add( self.lineObject )
32     def on_touch_move(self,t):
33         self.lineObject.points += (t.x,t.y)
34     def on_touch_up(self,t):
35         pass
36
37 #----- GUIの構築 -----
38 # ウィンドウの色とサイズ
39 Window.size = (600,400)
40 Window.clearcolor = (1,1,1,1)
41
42 # 最上位レイアウトの生成
43 root = BoxLayout(orientation='vertical')
44
45 # ボタンパネルの生成
46 btnpanel = BoxLayout(orientation='horizontal')
47 btnClear = BtnClear(text='Clear') # Clearボタンの生成
48 btnQuit = BtnQuit(text='Quit') # Quitボタンの生成
49 btnpanel.add_widget(btnClear) # Clearボタンの取り付け
50 btnpanel.add_widget(btnQuit) # Quitボタンの取り付け
51 btnpanel.size_hint = ( 1.0 , 0.1 ) # ボタンパネルのサイズ調整
52 root.add_widget(btnpanel) # ボタンパネルをメインウィジェットに取り付け
53
54 # 描画領域の生成
55 drawArea = DrawArea()
56 root.add_widget(drawArea)
```

```

57 |
58 | #----- アプリケーションの実行 -----
59 | ap = kivy03()
60 | ap.run()

```

全体の概要：

このプログラムの3～9行目で必要なモジュール群を読み込んでいる。必要となる各種のクラスの定義は13～35行目に記述している。GUIを構成するための各種のインスタンスの生成は43～56行目に記述しており、59～60行目にアプリケーションの実行を記述している。

BoxLayout へのウィジェットの登録

「Clear」「Quit」の2つのボタンは47～48行目で生成しており、これらを `add_widget` メソッドを用いて水平配置の `BoxLayout` である `btnpanel` に登録（49～50行目）している。同様の方法で、最上位の `BoxLayout`（垂直配置）である `root` に `btnpanel` と描画領域のウィジェット `drawArea` を登録（52, 56行目）している。

※ 親ウィジェットに子ウィジェットを登録、削除する方については「3.2.4 ウィジェットの登録と削除」（p.206）を参照のこと。

`BoxLayout` 配下に登録されたオブジェクトは均等大きさで配置されるが、今回の事例では、ボタンの領域の高さを小さく、描画領域の高さを大きく取っている。このように、配置領域の大きさの配分を変えるには、`BoxLayout` の子の要素に対して `size_hint` プロパティを指定する。51行目で実際にこれをしているが、親ウィジェットのサイズに対する比率を（水平比率, 垂直比率）の形で与える。

canvas に対する描画

Widget の `canvas` に対して描画するには `Graphics` クラスのオブジェクトを使用する。具体的には `Graphics` クラスのオブジェクトを `canvas` に対して `add` メソッドを用いて登録する。

`canvas` に対して登録できる `Graphics` オブジェクトには `Line`（折れ線）、`Rectangle`（長方形）、`Ellipse`（楕円）をはじめとする多くのものがある。また描画の色も `Color` オブジェクトを `canvas` に登録することで指定する。今回のプログラムでは29行目で `Color` オブジェクトを登録して描画色を指定している。また、30行目で作成した `Line` オブジェクトを31行目で登録し、33行目でこれを更新することで描画している。

イベント処理と描画の流れ

今回のプログラムでは、描画領域のオブジェクト `drawArea` に対するイベント処理によって描画を実現している。具体的には `DrawArea` クラスの定義の中で、タッチが開始（マウスボタンのクリックが開始）したことを受けるイベントハンドラである `on_touch_down` メソッド、ドラッグしたことを受けるイベントハンドラである `on_touch_move` メソッド、タッチが終了（マウスボタンが解放）したことを受けるイベントハンドラである `on_touch_up` メソッドを記述することで描画処理を実現している。これらメソッドは2つの仮引数を取る。

まず、`on_touch_down` で描画の開始をするが、`Color` オブジェクトの登録による色の指定（29行目）をして、`Line` オブジェクトを生成（30行目）して登録（31行目）している。このときはまだ `Line` オブジェクトは描画の開始点の座標のみを保持している。

次に、ドラッグが起こった際に `on_touch_move` で `Line` オブジェクトの座標を追加（33行目）することで、実際の描画を行う。タッチやマウスの座標はメソッドの第2引数である `t` に与えられるオブジェクトに保持されている。このオブジェクトの `x` プロパティに `x` 座標の値が、`y` プロパティに `y` 座標の値がある。

canvas の消去

`canvas` オブジェクトに対して `clear` メソッドを実行することでそこに登録された `Graphics` オブジェクトを全て消去する。今回のプログラムでは、消去ボタンである `btnClear` オブジェクトのタッチ（クリック）を受けるイベント処理でこのメソッドを実行（20行目）して描画を消去している。`Button` オブジェクトのタッチ（クリック）の開始と終了のイベントは、`on_press`, `on_release` メソッドで受ける。これらメソッドは1つの引数を取る。

アプリケーションの終了

今回のプログラムでは、終了ボタンである `btnQuit` オブジェクトに対するイベント処理でアプリケーションの終了を実現している。`sys` モジュールを読み込み、`exit` を呼び出すことでプログラムが終了する。

3.2.3 イベント処理（コールバックの登録による方法）

ウィジェットの拡張クラスを定義してイベントハンドリングのメソッドをオーバーライドする方法とは別に、bind メソッドを用いてイベントハンドリングする方法もある。

GUI のオブジェクトの生成後、そのオブジェクトに対して、

オブジェクト.bind(イベント=コールバック関数)

とすることで、bind メソッドの引数に指定したイベントが発生した際に、指定したコールバック関数を呼び出すことができる。この方法を採用することにより、イベント処理を目的とする拡張クラスの定義を省くことができる。

bind を用いてイベント処理を登録する形で先のプログラム kivy03.py を書き換えたプログラム kivy03-2.py を示す。

プログラム：kivy03-2.py

```
1  # coding: utf-8
2  import sys
3  from kivy.app import App
4  from kivy.uix.boxlayout import BoxLayout
5  from kivy.uix.button import Button
6  from kivy.uix.widget import Widget
7  from kivy.graphics import Color, Line
8  from kivy.core.window import Window
9
10 #----- 拡張クラスの定義 -----
11 # アプリケーションのクラス
12 class kivy03(App):
13     def build(self):
14         return root
15
16 #----- GUIの構築 -----
17 # ウィンドウの色とサイズ
18 Window.size = (600,400)
19 Window.clearcolor = (1,1,1,1)
20
21 # 最上位レイアウトの生成
22 root = BoxLayout(orientation='vertical')
23
24 # ボタンパネルの生成
25 btnpanel = BoxLayout(orientation='horizontal')
26 btnClear = Button(text='Clear') # Clearボタンの生成
27 btnQuit = Button(text='Quit') # Quitボタンの生成
28 btnpanel.add_widget(btnClear) # Clearボタンの取り付け
29 btnpanel.add_widget(btnQuit) # Quitボタンの取り付け
30 btnpanel.size_hint = ( 1.0 , 0.1 ) # ボタンパネルのサイズ調整
31 root.add_widget(btnpanel) # ボタンパネルをメインウィジェットに取り付け
32
33 # 描画領域の生成
34 drawArea = Widget()
35 root.add_widget(drawArea)
36
37 #----- コールバックの定義と登録 -----
38 # 描画領域を消去する関数
39 def callback_Clear(self):
40     drawArea.canvas.clear()
41     drawArea.canvas.add( Color(0.4,0.4,0.4,1) )
42     drawArea.lineObject = Line(points=[],width=10)
43     drawArea.canvas.add( drawArea.lineObject )
44 btnClear.bind(on_release=callback_Clear) # ボタンへの登録
45
46 # アプリケーションを終了する関数
47 def callback_Quit(self):
48     sys.exit()
49 btnQuit.bind(on_release=callback_Quit) # ボタンへの登録
50
51 # 描画のための関数
52 def callback_drawStart(self,t):
53     self.canvas.add( Color(0.4,0.4,0.4,1) )
54     self.lineObject = Line( points=(t.x,t.y), width=10 )
55     self.canvas.add( self.lineObject )
```

```

56 def callback_drawMove(self,t):
57     self.lineObject.points += (t.x,t.y)
58 def callback_drawEnd(self,t):
59     pass
60 drawArea.bind(on_touch_down=callback_drawStart)
61 drawArea.bind(on_touch_move=callback_drawMove)
62 drawArea.bind(on_touch_up=callback_drawEnd)
63
64 #----- アプリケーションの実行 -----
65 ap = kivy03()
66 ap.run()

```

全体の概要：

GUI 構築の部分は概ね kivy03.py と同じであるが、イベント処理のためのコールバック関数の定義と各ウィジェットへの登録が、39～62 行目に記述されている。

イベントハンドリングには、拡張クラスを定義してイベントハンドラをオーバーライドする方法と、bind メソッドによるコールバック関数の登録の 2 種類の方法があるが、プログラムの可読性を考慮してどちらの方法を採用するかを検討するのが良い。

3.2.4 ウィジェットの登録と削除

ウィジェットの階層的関係は、親の要素に子の要素を登録する方法で構築する。親のウィジェット wp に子のウィジェット wc を登録するには add_widget メソッドを用いて、

```
wp.add_widget(wc)
```

と実行する。また逆に wp から wc を削除するには remove_widget を用いて、

```
wp.remove_widget(wc)
```

と実行する。

次のプログラム kivy07.py は、一旦登録したウィジェットを取り除く処理を示すものである。

プログラム：kivy07.py

```

1  # coding: utf-8
2  from kivy.app import App
3  from kivy.uix.anchorlayout import AnchorLayout
4  from kivy.uix.button import Button
5  from kivy.core.window import Window
6
7  #----- 最上位のウィジェット -----
8  root = AnchorLayout()
9
10 #----- ボタンの生成 -----
11 btn1 = Button(text='Delete This Button!')
12
13 # 最上位のウィジェットからボタンを取り除く処理（コールバック関数）
14 def delbtn(self):
15     root.remove_widget(btn1)
16
17 # ボタンへのコールバックの登録
18 btn1.bind(on_release=delbtn)
19
20 #----- 最上位のウィジェットにボタンを登録 -----
21 root.add_widget(btn1)
22
23 #----- アプリケーションの実装 -----
24 class kivy07(App):
25     def build(self):
26         return root
27
28 Window.size=(250,50)
29 kivy07().run()

```

解説

このプログラムは最上位の `AnchorLayout` にボタンを1つ登録するものであり、そのボタンをクリック（タッチ）すると、そのボタン自身を取り除く。プログラムの実行例を図17に示す。

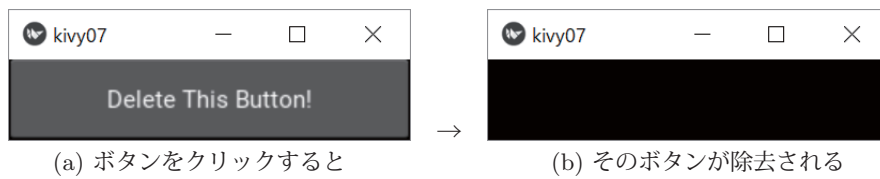


図 17: ウィジェット（ボタン）を取り除く処理

3.2.5 アプリケーションの開始と終了のハンドリング

Kivy アプリケーションの実行が開始する時と終了する時のハンドリングは、アプリケーションクラスの `on_start` メソッドと `on_stop` メソッドでそれぞれ行う。この様子を次のサンプルプログラム `kivy08.py` で確認できる。

プログラム：kivy08.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  from kivy.app import App
4  from kivy.uix.label import Label
5  from kivy.core.window import Window
6
7  #----- 最上位のウィジェット -----
8  root = Label(text='This is a sample.')
9
10 #----- アプリケーションの実装 -----
11 class kivy08(App):
12
13     # アプリケーションのインスタンス生成
14     def build(self):
15         print('0) アプリケーションのインスタンスが生成されました. ')
16         return root
17
18     # アプリケーション実行開始時
19     def on_start(self):
20         print('1) アプリケーションの実行が開始されました. ')
21
22     # アプリケーション終了時
23     def on_stop(self):
24         print('2) アプリケーションを終了します. ')
25
26 Window.size=(250,50)
27 kivy08().run()
```

このプログラムを実行すると図18のようなウィンドウが表示される。

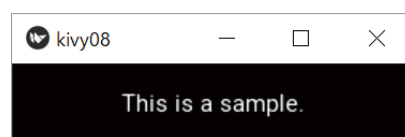


図 18: Kivy08.py のアプリケーションウィンドウ

アプリケーションの生成時、開始時、終了時にそれぞれメッセージを標準出力に出力する。（次の例参照）

- | | |
|-----------------------------|------|
| 0) アプリケーションのインスタンスが生成されました. | ←生成時 |
| 1) アプリケーションの実行が開始されました. | ←開始時 |
| 2) アプリケーションを終了します. | ←終了時 |

3.3 各種ウィジェットの使い方

ここでは使用頻度の高い代表的なウィジェットの基本的な使い方について説明する。より詳細な使用方法については Kivy の公式サイトを参照のこと。またここで紹介するウィジェット以外にも有用なものが多くあり、それらについて

ても公式サイトを参照のこと。

3.3.1 ラベル：Label

GUI に文字を表示する場合に標準的に用いられるのがラベル (Label) オブジェクトである。Label のインスタンスを生成する際、コンストラクタの引数に `text='文字列'` とすると、与えた文字列を表示するラベルが生成される。図 19 は `Label(text='This is a test for Label object.')` として生成したラベルを表示した例である。



図 19: Label オブジェクトの表示

ラベルに表示する文字のフォントを指定することができる。特に本書を執筆中の版の Kivy は、フォントを指定せずに日本語の文字列を表示することができないので、日本語文字列を表示する場合にはフォント指定が必須である。図 20 は Windows 環境で

```
Label( text='日本語を MS ゴシックで表示するテスト',  
      font_name=r'C:\Windows\Fonts\msgothic.ttc',  
      font_size='24pt')
```

として、MS ゴシックフォントを使用して表示した例である。この書き方の中にある `font_name` はフォントのパス¹⁷²を、`font_size` はフォントのサイズを指定するものである。



図 20: Label オブジェクトの表示 (MS ゴシックによる日本語表示)

使用するフォントによっては、Kivy 独自のマークアップを使用して、強調、斜体などのスタイルを施すことができる。マークアップは与える文字列の中に直接記述できる。図 21 は、

```
Label( text='[u] 日本語を IPA 明朝で表示するテスト (下線付き) [/u]',  
      markup=True, color=(0.6,1,1,1), font_name=r'C:\Windows\Fonts\ipam.ttf',  
      font_size='20pt')
```

として、IPA 明朝フォントを使用して下線 (アンダーライン) を施した例である。

マークアップを使用する場合は `markup=True` を指定する。またこの例のようにフォントの色を指定する際は

`color=(R,G,B, α)`

と指定する。

マークアップは `'[...]~[/...]'` で括るタグを使用する。代表的なマークアップには

- [b] 文字列 [/b] 強調文字
- [u] 文字列 [/u] 下線 (アンダーライン)
- [i] 文字列 [/i] 斜体 (イタリック)

といったものがある。




図 21: Label オブジェクトの表示 (IPA 明朝による日本語表示：下線付き)

Label のコンストラクタに与える代表的なキーワード引数：

`text='文字列'`

`font_name=r'フォントのパス'`

`font_size='フォントサイズ'`

`color=[赤, 緑, 青, α (不透明度)]` 全て 0~1 の値

`markup=True` マークアップを使用する場合

¹⁷²フォントのパスは OS 毎に異なるので注意すること。

ここで示した Label を表示するサンプルプログラムを kivy02Label.py に示す.

プログラム: kivy02Label.py

```
1 # coding: utf-8
2 #----- モジュールの読み込み -----
3 from kivy.app import App
4 from kivy.uix.boxlayout import BoxLayout
5 from kivy.uix.label import Label
6 from kivy.core.window import Window
7
8 #----- クラスの定義 -----
9 # アプリケーションのクラス
10 class kivy02(App):
11     def build(self):
12         return root
13
14 #----- GUIの構築 -----
15 # 最上位のレイアウト
16 root = BoxLayout(orientation='vertical')
17
18 # ラベル
19 lb1 = Label(
20     text='This is a test for Label object.',
21     font_size='24pt')
22 lb2 = Label(
23     text='日本語をMSゴシックで表示するテスト',
24     font_name=r'C:\Windows\Fonts\msgothic.ttc',
25     font_size='24pt')
26 lb3 = Label(
27     text='日本語をMS明朝で表示するテスト',
28     font_name=r'C:\Windows\Fonts\msmincho.ttc',
29     font_size='24pt')
30 lb4 = Label(
31     text='日本語をIPAゴシックで表示するテスト',
32     color=(1,1,0,1),
33     font_name=r'.\ipag.ttf',
34     font_size='24pt')
35 lb5 = Label(
36     text='[u]日本語をIPA明朝で表示するテスト（下線付き）[/u]',
37     markup=True,
38     color=(0.6,1,1,1),
39     font_name=r'.\ipam.ttf',
40     font_size='20pt')
41 root.add_widget(lb1)
42 root.add_widget(lb2)
43 root.add_widget(lb3)
44 root.add_widget(lb4)
45 root.add_widget(lb5)
46
47 # ウィンドウの色とサイズ
48 Window.size = (640,300)
49
50 #----- アプリケーションの起動 -----
51 ap = kivy02()
52 ap.run()
```

注意) このプログラムを実行する場合は、フォントのパスなどを適切に編集すること。

3.3.1.1 リソースへのフォントの登録

フォントファイルが収められているディレクトリのパスを Kivy のリソース (resource) に登録しておくと font_name の指定において、フォントのファイル名のみで記述で済む。フォントファイルのディレクトリをリソースに登録するには、resource_add_path メソッドを使用する。このメソッドを使用するためには次のようにして必要なモジュールを読み込んでおく。

```
from kivy.resources import resource_add_path
```

その後 resource_add_path メソッドを実行する。

《 フォントパスのリソースへの登録 》

`resource_add_path(フォントディレクトリのパス)`

例えば Windows 環境では、

```
resource_add_path(r'C:\Windows\Fonts')
```

などとする。

■ フォントファイルと格納場所に関すること

本書では Windows 環境を前提とした解説を基本としており、Label オブジェクトのためのフォントファイルも `C:\Windows\Fonts` 配下にあることを前提としている。しかし、クロスプラットフォームに対応したアプリケーションプログラムを開発する場合は、各 OS 固有のフォントディレクトリを前提とするのは好ましくない。また、各 OS に強く関連づけられたフォント（Windows における MS ゴシックや MS 明朝など）を使用するのもあまり好ましくない。従って、クロスプラットフォームで動作するアプリケーションを開発する場合は、IPA¹⁷³ や Google などが公開する制約条件の緩いライセンスのフォントなどをアプリケーション独自のディレクトリに配置して使用する方が良い。

■ デフォルトフォントの設定

DEFAULT_FONT の設定をしておくと、font_name の設定を省略した際のデフォルトフォントを指定できる。デフォルトフォントの設定には LabelBase クラスの register メソッドを使用する。これを行うには次のようにして必要なモジュールを読み込んでおく。

```
from kivy.core.text import LabelBase, DEFAULT_FONT
```

この後デフォルトフォントを設定する。

《 デフォルトフォントの設定 》

`LabelBase.register(DEFAULT_FONT, フォントファイル名)`

例えば Windows 環境で IPA ゴシックフォントをデフォルトフォントにするには、

```
LabelBase.register(DEFAULT_FONT, 'ipag.ttf')
```

などとする。

3.3.2 ボタン：Button

Button はボタンを実現するクラスである。ボタントップに表示する文字列の扱いについては Label の場合とほぼ同じであるが、下記のようにボタンのタッチ（クリック）によるイベント処理ができる点が特徴である。

タッチ（クリック）のイベント：

```
on_press    タッチ（クリック）の開始
on_release  タッチ（クリック）の終了
```

これらのイベントハンドリングには引数が 1 つ（自オブジェクト：self）与えられる。

bind メソッドでコールバック関数を登録する場合は、

```
ボタンオブジェクト.bind(on_press=コールバック関数)
ボタンオブジェクト.bind(on_release=コールバック関数)
```

とする。コールバック関数の定義を記述する際も仮引数は 1 つである。

「ボタンがクリックされた」ことをハンドリングするには「ボタンが放された」と解釈して on_release でハンドリングするとよい。

3.3.3 テキスト入力：TextInput

TextInput は文字列の入力、編集をする場合に使用する。フォントやフォントサイズの設定は Label の場合と同じである。

¹⁷³ 情報処理推進機構 (<https://www.ipa.go.jp/>)

テキストの入力や変更の際に起こるイベント

TextInput オブジェクト内でテキストの新規入力や変更があった場合、それがイベントとして発生する。TextInput の拡張クラスを定義する際はそれを `on_text` メソッドとしてハンドリングする。また TextInput オブジェクトに対して `bind` メソッドでコールバック関数を登録する際は

オブジェクト.bind(text=コールバック関数)

とする。 `on_text` メソッドでイベントハンドリングする際は仮引数を 3 つ取り、コールバック関数でハンドリングする際は仮引数を 2 つ取る。両方の場合において、第 1 引数には、そのオブジェクト自身 (self) が与えられる。

入力されているテキストの文字列は、プロパティ `text` に保持されている。

複数行にまたがるテキストを扱うかどうかに関しては、コンストラクタにキーワード引数 '`multiline=True/False`' を与えることで設定する。 `True` を与えると複数行の取り扱い、 `False` を与えると 1 行のみの取り扱いとなる。

3.3.4 チェックボックス：CheckBox

チェックボックス (CheckBox オブジェクト) のイベントハンドリングは基本的にボタンと同様 (`on_press`, `on_release`) である。チェックされているか否かはチェックボックスオブジェクトのプロパティ `active` から真理値 (`True/False`) として得られる。

3.3.5 進捗バー：ProgressBar

進捗バー (ProgressBar オブジェクト) は値の大きさを水平方向に可視化するものである。値のプロパティは `value` である。可視化範囲の最大値はプロパティ `max` に設定する。

3.3.6 スライダ：Slider

スライダ (Slider オブジェクト) は縦あるいは横方向にスライドするウィジェットであり、視覚的に値を調整、入力する際に用いる。値のプロパティは `value` である。このクラスのインスタンスを生成する際、キーワード引数 `orientation=` を与えることで、縦横のスタイルを選択できる。この引数の値に '`horizontal`' を与えると横方向、 '`vertical`' を与えると縦方向になる。(デフォルトは水平)

スライダの値の変更に伴って起こるイベント

Slider オブジェクトを操作 (値を変更) した場合、それがイベントとして発生する。Slider の拡張クラスを定義する際はそれを `on_value` メソッドとしてハンドリングする。また Slider オブジェクトに対して `bind` メソッドでコールバック関数を登録する際は

オブジェクト.bind(value=コールバック関数)

とする。 `on_value` メソッドでイベントハンドリングする際は仮引数を 3 つ取り、コールバック関数でハンドリングする際は仮引数を 2 つ取る。両方の場合において、第 1 引数には、そのオブジェクト自身 (self) が与えられる。

3.3.7 スイッチ：Switch

スイッチ (Switch オブジェクト) は水平方向の「切り替えスイッチ」で ON/OFF の 2 つの状態を取る。それぞれの状態はプロパティ `active` から真理値 (`True/False`) として得られる。(デフォルトは OFF)

スイッチの切り替えに伴って起こるイベント

Switch オブジェクトを操作 (値を変更) した場合、それがイベントとして発生する。Switch の拡張クラスを定義する際はそれを `on_active` メソッドとしてハンドリングする。また Switch オブジェクトに対して `bind` メソッドでコールバック関数を登録する際は

オブジェクト.bind(active=コールバック関数)

とする。 `on_active` メソッドでイベントハンドリングする際は仮引数を 3 つ取り、コールバック関数でハンドリングする際は仮引数を 2 つ取る。両方の場合において、第 1 引数には、そのオブジェクト自身 (self) が与えられる。

3.3.8 トグルボタン：ToggleButton

トグルボタン (ToggleButton オブジェクト) はいわゆる「ラジオボタン」と同様のものである。すなわち、複数のボタンを1つの「グループ」としてまとめ、同一のグループ内のボタンの内、1つだけが ON (チェック済みもしくはダウン) になるウィジェットである。イベントハンドリングは基本的に Button クラスと同様であるが、プロパティ state に押されていない状態を意味する 'normal' か、押されている状態を意味する 'down' が保持されている。

トグルボタンの例

次のように3つのトグルボタン tb1, tb2, tb3 を生成した例について考える。

```
tb1 = ToggleButton(group='person',text='nakamura',state='down')
tb2 = ToggleButton(group='person',text='tanaka')
tb3 = ToggleButton(group='person',text='itoh')
```

これらを BoxLayout で水平に配置した例が図 22 である。



図 22: トグルボタンの例

トグルボタン生成時のコンストラクタに、キーワード引数 group= を与えることで複数のトグルボタンをグループ化することができる。またグループ内の1つのトグルボタンに state='down' を指定することで、初期状態で押されているトグルボタンを決めることができる。

3.3.9 画像：Image

Image は画像を扱うためのウィジェットクラスである。そのインスタンスに画像ファイルを割当ててすることで画像を配置することができる。インスタンス生成時のコンストラクタにキーワード引数 source= を指定することで画像ファイルを割り当てる。

《 Image オブジェクト 》

`Image(source=r' 画像ファイルのパス')`

指定したファイルから画像を読み込む。

Image オブジェクトの texture_size プロパティに読み込んだ画像のピクセルサイズが保持されている。

画像の表示サイズに関しては、それを配置するレイアウトオブジェクトに制御を委ねるのが一般的である。次に示すサンプルプログラムは、スライダに連動して画像の表示サイズが変化するものである。

3.3.9.1 サンプルプログラム

スライダの値によって画像の表示サイズが変わるプログラム kivy02Earth.py を示す。拡大、縮小する画像を常に中央に表示するため、Image オブジェクトを AnchorLayout で配置している。

プログラム：kivy02Earth.py

```
1 # coding: utf-8
2 #----- モジュールの読み込み -----
3 from kivy.app import App
4 from kivy.uix.boxlayout import BoxLayout
5 from kivy.uix.anchorlayout import AnchorLayout
6 from kivy.uix.image import Image
7 from kivy.uix.slider import Slider
8 from kivy.core.window import Window
9
10 #----- GUIとアプリケーションの定義 -----
11 # イベントのコールバック
12 def onValueChange(self,v):
13     im.size_hint = ( sl.value, sl.value )
14
15 # GUIの構築
```

```

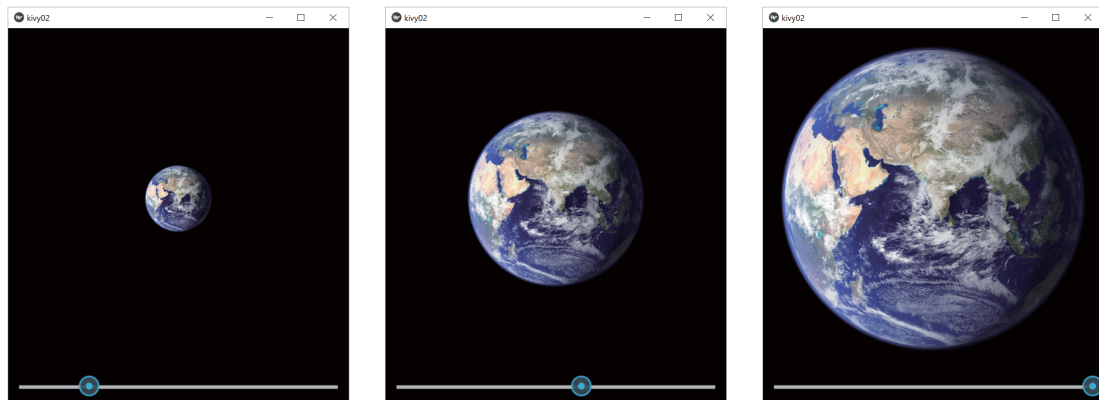
16 root = BoxLayout(orientation='vertical')
17
18 # 画像とそのレイアウト
19 anchor = AnchorLayout()
20 im = Image(source='Earth.jpg')
21 anchor.add_widget(im)
22
23 # スライダー
24 sl = Slider(min=0.0,max=1.0,step=0.01)
25 sl.size_hint = ( 1.0 , 0.1 )
26 sl.value = 1.0
27 sl.bind(value=onValueChange)
28
29 root.add_widget(anchor)
30 root.add_widget(sl)
31
32 # アプリケーションのクラス
33 class kivy02(App):
34     def build(self):
35         return root
36
37 # ウィンドウの色とサイズ
38 Window.size = (500,550)
39
40 # アプリ起動
41 print(im.texture_size) # 画像サイズの調査
42 ap = kivy02()
43 ap.run()

```

全体の概要：

最上位のレイアウト（垂直の BoxLayout）である root は、画像表示部とスライダーを収めるものである。Image オブジェクト im は AnchorLayout オブジェクト anchor に収められており、その内部で im の size_hint を調整することで表示サイズを調節している。

このプログラムを実行した様子を図 23 に示す。



スライダーに連動して画像の表示サイズが変わる

図 23: kivy02Earth.py の実行例

3.4 Canvas グラフィックス

ウィジェット（Widget）には canvas 要素があり、これに対してグラフィックスの描画ができる。先の「3.2.2 アプリケーション構築の例」（p.202）でも少し説明した通り、canvas に対して色や図形などを追加することで描画ができる。

Widget を使用するには、必要なモジュールを、

```
from kivy.uix.widget import Widget
```

として読み込み、これの canvas に対して Graphics を描画する。基本的な Graphics オブジェクトには Color, Line, Rectangle, Ellipse があり、それらを使用する際は kivy.graphics からインポートする。

Graphics モジュールを読み込む例

```
from kivy.graphics import Color, Line, Rectangle, Ellipse
```


こうすることで、複数の Graphics のモジュールを読み込むことができる。

3.4.1 Graphics クラス

以下に紹介する Graphics オブジェクトを canvas に対して add メソッドで登録して描画する。
Kivy の座標系は他の多くの GUI ライブラリと異なり、上下の方向が逆である。(図 24)

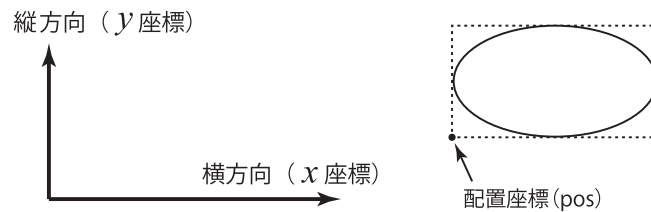


図 24: Kivy の座標系

また Kivy 以外の多くの GUI ライブラリはオブジェクトの配置を決める際、オブジェクトの左上の位置を指定するが、Kivy では Graphics オブジェクトの左下の位置を配置座標 (pos の値) とする。

3.4.1.1 Color

描画色を指定するためのオブジェクトのクラスである。

コンストラクタ: `Color(Red, Green, Blue, Alpha)`

引数は全て 0~1.0 の範囲の値である。Alpha は不透明度を指定するもので、1.0 を指定すると、完全に不透明になる。

3.4.1.2 Line

折れ線を描画するためのオブジェクトのクラスである。

コンストラクタ: `Line(points=座標リスト, width=線幅)`

canvas 上の座標, $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ を結ぶ折れ線を、線の幅 width で描画する。描画座標リストは $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$ とする。

3.4.1.3 Rectangle

長方形領域を描画するためのオブジェクトのクラスである。canvas 上に画像を表示する場合にも Rectangle を使用する。

コンストラクタ 1: `Rectangle(pos=(描画位置の座標), size=(横幅, 高さ))`

canvas 上の pos で指定した座標に、size で指定したサイズの長方形を描く。

コンストラクタ 1: `Rectangle(pos=(描画位置の座標), size=(横幅, 高さ),
texture=テクスチャオブジェクト)`

canvas 上の pos で指定した座標に、size で指定したサイズでテクスチャオブジェクトを描く。

テクスチャは canvas に画像を表示する際の標準的なオブジェクトであり、これの使用に際しては、下記のようにして必要なモジュールを読み込んでおく。

```
from kivy.graphics.texture import Texture
```

先に、「3.3.9 画像: Image」(p.212) で画像を読み込んで保持するウィジェットである Image について説明したが、テクスチャオブジェクトはこの Image オブジェクトから texture プロパティとして取り出すことができる。

3.4.1.4 Ellipse

楕円を描画するためのオブジェクトのクラスである。

コンストラクタ： `Ellipse(pos=(描画位置の座標), size=(横幅, 高さ))`
canvas 上の pos で指定した座標に, size で指定したサイズの楕円を描く。

この他にも Bezier オブジェクトもあり, 多角形や曲線を描くことができる,

3.4.2 サンプルプログラム

3.4.2.1 正弦関数のプロット

canvas グラフィックスを使うと, 簡単に数学関数の軌跡がプロットできる。

プログラム：kivy04-1.py

```
1 # coding: utf-8
2 #----- 必要なパッケージの読み込み -----
3 import math
4 from kivy.app import App
5 from kivy.uix.widget import Widget
6 from kivy.graphics import Color, Rectangle
7 from kivy.core.window import Window
8
9 #----- アプリケーションの構築 -----
10 class kivy04(App):
11     def build(self):
12         return root
13
14 root = Widget()
15
16 root.canvas.add( Color(1,0,0,1) )
17 x = 0.0
18 while x < 6.28:
19     y = 100.0*math.sin(x)
20     root.canvas.add( Rectangle( pos=(32.0*x+5.0,y+105.0), size=(3,3) ) )
21     x += 0.005
22
23 # アプリの実行
24 Window.size = (210,210)
25 kivy04().run()
```

このプログラムの実行結果を図 25 に示す。

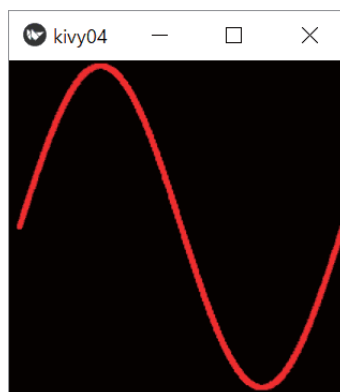


図 25: kivi04-1.py の実行結果

3.4.2.2 各種図形, 画像の表示

長方形, 楕円, 折れ線, テクスチャ画像を表示するプログラム kivy04-2.py を示す。

プログラム：kivy04-2.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  from kivy.app import App
4  from kivy.uix.widget import Widget
5  from kivy.uix.image import Image
6  from kivy.graphics import Color, Line, Rectangle, Ellipse
7  from kivy.core.window import Window
8
9  #----- アプリケーションの構築 -----
10 class kivy04(App):
11     def build(self):
12         return root
13
14 # 画像の読み込みとテクスチャの取り出し
15 im = Image(source='Earth.jpg');          tx = im.texture
16
17 root = Widget()
18 # 長方形の描画
19 root.canvas.add( Color(1,0,0,1) )
20 root.canvas.add( Rectangle(pos=(10,210),size=(350,100)) )
21 # 画像の描画
22 root.canvas.add( Color(1,1,1,1) )
23 root.canvas.add( Rectangle(pos=(0,0),texture=tx,size=(210,200)) )
24 # 楕円の描画
25 root.canvas.add( Color(0,1,0,1) )
26 root.canvas.add( Ellipse(pos=(200,10),size=(170,170)) )
27 # 折れ線の描画
28 root.canvas.add( Color(1,1,1,1) )
29 root.canvas.add( Line(width=10,points=[580,25,400,25,580,160,400,290,580,290]) )
30
31 # スクリーンショット
32 def save_shot(self,t):
33     # 方法-1
34     Window.screenshot(name='kivy04-2-1.png')
35     # 方法-2
36     self.export_to_png('kivy04-2-2.png')
37
38 root.bind(on_touch_up=save_shot)
39
40 # アプリの実行
41 Window.size = (600,330)
42 kivy04().run()
```

このプログラムの実行結果を図 26 に示す。



図 26: kivi04-2.py の実行結果

プログラムの 32～36 行目で Window のスクリーンショットの画像を保存する機能を実装している。

```
Window.screenshot(name='kivy04-2-1.png')
```

としている部分がスクリーンショットのメソッドの 1 つ (screenshot メソッド) であり、その時点のウィンドウの様子を引数 'name=' に与えたファイル名の png 形式画像として保存する。(保存時のファイル名には自動的に連番が付

けられる) また,

```
self.export_to_png('kivy04-2-2.png')
```

としている部分も同じ処理を実現するもので、これは Widget の表示内容を PNG 形式の画像として保存するものである。(Widget に対する export_to_png メソッド)

3.4.3 フレームバッファへの描画

Canvas への描画とは別に、**フレームバッファ (FBO)**¹⁷⁴ と呼ばれる固定サイズの領域への描画が可能である。フレームバッファからは**ピクセル値の取り出し**が可能である。

FBO 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んでおく。

```
from kivy.graphics import Fbo
```

FBO の生成時には、描画サイズを指定する。

例. FBO の生成

```
f = Fbo( size=(400,300) )
```

これで画素サイズ 400 × 300 の FBO が f として生成される。FBO は描画対象の Widget の Canvas に登録しておく必要がある。

例. Widget オブジェクト root への FBO オブジェクト f の登録

```
root.canvas.add( f )
```

更にこの後、FBO の texture プロパティを与えた Rectangle を Canvas に描画することで実際に FBO の内容が表示される。

FBO への描画は Canvas への描画とほぼ同じ方法 (add メソッド) が使用できる。

FBO を用いて、タッチした場所の画素の値 (ピクセル値) を取得するプログラム kivy04-3.py を次に示す。

プログラム: kivy04-3.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読み込みとモジュールの初期設定 -----
3  from kivy.config import Config
4  Config.set('graphics','resizable',False)      # ウィンドウサイズの禁止
5  from kivy.app import App
6  from kivy.uix.widget import Widget
7  from kivy.graphics import Color, Rectangle, Fbo
8  from kivy.core.window import Window
9
10 #----- アプリケーションの構築 -----
11 class kivy04(App):
12     def build(self):
13         return root
14
15 # 最上位ウィジェットの生成
16 root = Widget()
17
18 # フレームバッファの生成とCanvasへの登録
19 fb = Fbo(size=(300, 150))
20 root.canvas.add( fb )
21
22 # フレームバッファへの描画
23 fb.add( Color(1, 0, 0, 1) )
24 fb.add( Rectangle(pos=(0,0),size=(100, 150)) )
25 fb.add( Color(0, 1, 0, 1) )
26 fb.add( Rectangle(pos=(100,0),size=(100, 150)) )
27 fb.add( Color(0, 0, 1, 1) )
28 fb.add( Rectangle(pos=(200,0),size=(200, 150)) )
29 fb.add( Color(1,1,1,1) )
30 fb.add( Rectangle(pos=(0,75),size=(300,75)) )
31 # フレームバッファの内容をCanvasに描画
32 root.canvas.add( Rectangle(size=(300, 150), texture=fb.texture) )
```

¹⁷⁴OpenGL の描画フレーム

```

33
34 # タッチ位置の色の取得（コールバック関数）
35 def pickColor(self,t):
36     # ウィンドウサイズの取得
37     (w,h) = Window.size
38     # タッチ座標の取得
39     (x,y) = t.spos
40     # フレームバッファ上での座標
41     fbx = int(w*x);    fby = int(h*y)
42     # フレームバッファ上のピクセル値の取得
43     c = fb.get_pixel_color(fbx,fby)
44     print( '位置:\t',(fbx,fby),'\tピクセル:',c )
45
46 # コールバックの登録
47 root.bind(on_touch_up=pickColor)
48
49 # アプリの実行
50 Window.size = (300,150)
51 kivy04().run()

```

解説.

19～20行目でFBOを生成してWidgetのCanvasに登録している。23～30行目でFBOに対して描画し、それを32行目でRectangleとしてCanvasに描画している。

アプリケーションのウィンドウ内をタッチ（クリック）するとコールバック関数 pickColor が呼び出され、その位置のピクセルを43行目で取得している。

3.4.3.1 ピクセル値の取り出し

FBOのピクセル値を取り出すには get_pixel_color メソッドを使用する。

書き方： FBOオブジェクト.get_pixel_color(横位置, 縦位置)

ピクセル値を取り出す横位置と縦位置は、画像左下を基準とするピクセル位置である。得られたピクセル値は

[赤, 緑, 青, α]

のリストであり、各要素は 0～255 の整数値である。

プログラム kivy04-3.py を実行すると図 27 のようなウィンドウが表示される。

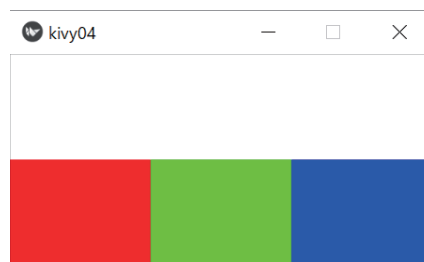


図 27: kivy04-3.py の実行結果

このウィンドウ内をタッチ（クリック）すると、次の例のように、その位置のピクセル値を表示する。

```

位置: (143, 117)   ピクセル: [255, 255, 255, 255]
位置: (52, 39)    ピクセル: [255, 0, 0, 255]
位置: (134, 34)   ピクセル: [0, 255, 0, 255]
位置: (256, 24)   ピクセル: [0, 0, 255, 255]

```

3.4.3.2 イベントから得られる座標位置

Kivy のウィジェット上のイベントから取得される座標位置は、ウィジェットサイズの縦横を共に 1.0 に正規化した位置であり、基準は左下である。このため、先のプログラム kivy04-3.py では、37～41 行目にあるように FBO 上の座標位置を得るための変換処理をしている。

3.5 スクロールビュー (ScrollView)

大きなサイズのウィジェットやレイアウトを、それよりも小さなウィジェットやウィンドウの内部でスクロール表示する場合はスクロールビュー (ScrollView) を使用する。これはスクロールバーを装備した矩形領域であり、内部のオブジェクト (子要素) を縦横に並行移動して表示するものである。

スクロールビューの使用に際して、次のようにして必要なモジュールを読み込む。

```
from kivy.uix.scrollview import ScrollView
```

スクロールビューは次のようにしてインスタンスを生成し、基本的にはウィジェットの1つとして扱う。

コンストラクタ: `ScrollView()`

コンストラクタにキーワード引数「`bar.width=幅`」を与えることで、スクロールバーの幅を設定することができる。

ここではサンプルプログラムを示しながらスクロールビューの使用方法について説明する。

【サンプルプログラム】

図 28 のような、縦横にたくさんのボタンが配置されたウィジェットをスクロール表示するアプリケーションプログラム `kivy06.py` を考える。

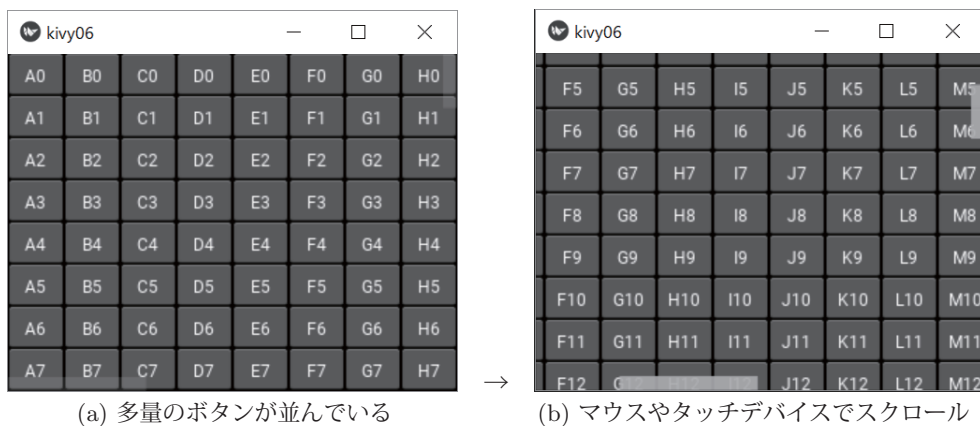


図 28: 縦横に並んだボタンパネルのスクロール

基本的な考え方:

スクロールビューよりも大きなサイズのウィジェットを、スクロールビューに子要素として登録する。次に示すプログラム `kivy06.py` では、多量のボタンを配置した巨大な `BoxLayout` を生成して、それをスクロールビューに登録している。

プログラム: `kivy06.py`

```
1 # coding: utf-8
2 ##### 必要なパッケージの読み込み #####
3 from kivy.app import App
4 from kivy.uix.boxlayout import BoxLayout
5 from kivy.uix.button import Button
6 from kivy.uix.scrollview import ScrollView
7 from kivy.core.window import Window
8 ##### GUIの構築 #####
9 # ウィンドウのサイズ
10 Window.size = (320,240)
11
12 # 最上位ウィジェット (スクロールビュー) の生成
13 root = ScrollView( bar_width=10 )
14
15 # 大きいボックスレイアウトの作成
16 bx = BoxLayout( orientation='vertical',
17                 size_hint=(None, None),      # サイズに関して親の制御を受けない設定
18                 size=(1040,1500) )          # 1040×1500の固定サイズ
19
20 # A0~Z49 のボタンを生成 (1300個)
21 al = [chr(i) for i in range(65, 65+26)]      # アルファベットのリスト
22 # 0~49行のボタン配列を生成
```



```

23 for m in range(50):
24     bx2 = BoxLayout( orientation='horizontal' )
25     # An～Znの横方向のボタン生成 (26個)
26     for n in al:
27         # 40×30のサイズのボタンを生成
28         bx2.add_widget( Button(text=n+str(m), font_size=12,
29                               size_hint=(None, None), size=(40,30) ) )
30     bx.add_widget(bx2)
31 root.add_widget(bx)
32 ##### アプリケーションの構築 #####
33 # アプリケーションのクラス
34 class kivy06(App):
35     def build(self):
36         return root
37
38 # アプリケーションの実行
39 kivy06().run()

```

解説

16～18行目で生成した大きなサイズの BoxLayout である bx に多量のボタンを配置して、それを、13行目で生成したスクロールビュー root に31行目で登録している。

23～30行目は多量のボタンを生成している部分である。An～Zn (n は整数値) の26個のボタンを生成して、それらを水平方向 (行) の BoxLayout である bx2 に登録して、それを垂直方向の BoxLayout である bx に次々と登録している。

3.5.1 ウィジェットのサイズ設定

通常の場合は、階層的に構築されたウィジェット群のサイズは自動的に調整される。これは、親ウィジェットに収まるように子ウィジェットのサイズを調整するという Kivy の機能によるものであるが、先のプログラム kivy06.py では、ウィジェットサイズの自動調整の機能に任せることなく、大きなサイズのウィジェット (1,040 × 1,500 のサイズの BoxLayout) を生成している。これは、ウィジェット生成時のコンストラクタにキーワード引数

```
size_hint=(None, None)
```

を与えることで可能となる。これと同時に、コンストラクタにキーワード引数

```
size=(横幅, 高さ)
```

を与えて、具体的なサイズを設定する。

3.5.2 マウスのドラッグによるスクロール

ScrollView のスクロールバーをマウスのドラッグで操作するには ScrollView のコンストラクタにキーワード引数

```
scroll_type=['bars']
```

を与える。

3.6 ウィンドウサイズを固定 (リサイズを禁止) する設定

ウィンドウのリサイズを禁止 (ウィンドウサイズを固定) するには、Kivy モジュールを読み込む先頭の位置で、次のように設定する。

```

from kivy.config import Config
Config.set('graphics', 'resizable', False)

```

これは Kivy の他のモジュールの読み込みに先立って記述すること。

3.7 Kivy 言語による UI の構築

実用的な GUI アプリケーションを構築する場合、GUI の構築と編集の作業に多くの時間と労力を要する。この部分の作業を容易にするために、インターフェースを構築するための特別な機能を使用することが、アプリケーション開

発において一般的になって¹⁷⁵ きている。Python と Kivy によるアプリケーション開発においても、インターフェース構築に特化した言語「**Kivy 言語**」（以下 Kv と略す）を使用することができ、開発効率を高めることができる。

Kv は Python とは異なる独自の言語であり、ここでは Kv 自体の基礎的な解説からはじめ、Kv で記述した GUI と Python で記述したプログラムとの相互関係について説明する。ただし本書は Kv の全般的なリファレンスではなく、あくまで導入的な内容に留める。

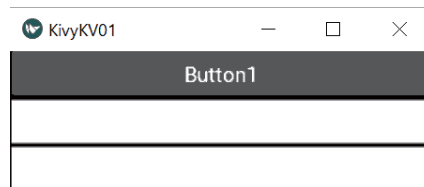
3.7.1 Kivy 言語の基礎

Kv はレイアウトやウィジェットの階層構造である**ウィジェットツリー**を宣言的に記述する言語である。Kv で記述されたウィジェットツリーは可読性が高く、GUI の構造全体の把握が容易になる。

3.7.1.1 サンプルプログラムを用いた説明

まずは Kv を使用せずに GUI を構築したアプリケーションを示す。そして、同じ機能を持つアプリケーションの GUI を Kv で記述する例を示す。

サンプルとして示すアプリケーションは図 29 に示すようなものである。これは、1 段目にボタン、2 段目と 3 段目にテキスト入力を備えたもので、それらを BoxLayout で配置している。



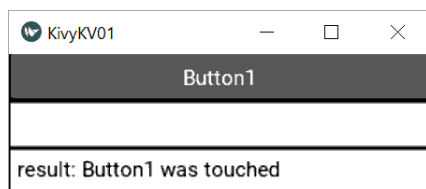
1 段目がボタン、2 段目と 3 段目がテキスト入力

図 29: サンプルプログラムの実行例（起動時）

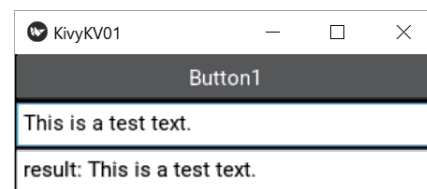
このアプリケーションの動作は次のようなものである。

- (a) ボタンをタッチ（クリック）すると、3 段目にその旨のメッセージが表示される。
- (b) 2 段目にテキストを入力すると、3 段目にその旨のメッセージが表示される。

これら動作の様子を図 30 に示す。



(a) ボタンをクリックしたときの反応



(b) 2 段目にテキストを入力したときの反応

図 30: サンプルプログラムの実行例（動作）

このアプリケーションを Kv を使用せずに構築したものがプログラム kivyKV01-1.py である。

プログラム：kivyKV01-1.py

```
1 # coding: utf-8
2 #----- 関連モジュールの読み込み -----
3 from kivy.app import App
4 from kivy.uix.boxlayout import BoxLayout
5 from kivy.uix.button import Button
6 from kivy.uix.textinput import TextInput
7 from kivy.core.window import Window
8
9 #----- GUIの構築 -----
10 # コールバック関数
11 def funcButton(self):
12     tx2.text = 'result: Button1 was touched'
13 def funcTextInput(self, v):
```

¹⁷⁵JavaFX における FXML や、それらを基本とする統合開発環境はまさにその例である。

```

14         tx2.text = 'result: ' + self.text
15
16 # GUI
17 root = BoxLayout(orientation='vertical')
18 bt1 = Button(text='Button1')
19 bt1.bind(on_release=funcButton)
20 tx1 = TextInput()
21 tx1.bind(text=funcTextInput)
22 tx2 = TextInput()
23 root.add_widget(bt1)
24 root.add_widget(tx1)
25 root.add_widget(tx2)
26
27 #----- アプリケーション本体 -----
28 class KivyKV01App(App):
29     def build(self):
30         return root
31
32 # アプリの実行
33 Window.size = (300,100)
34 KivyKV01App().run()

```

次に、このプログラムのインターフェース部を Kv で、その他を Python で記述することを考える。このアプリケーションの GUI の構成を階層的に示すと図 31 のようになる。

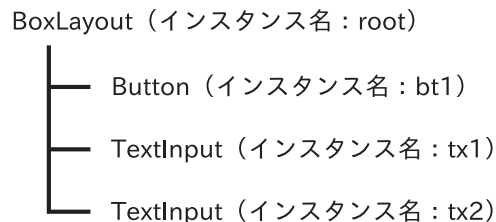


図 31: サンプルプログラムのウィジェットツリー

Kv で GUI を記述するとこの階層構造をそのまま反映した形となる。その Kv ファイル kivyKV01.kv を示す。

Kv ファイル: kivyKV01.kv

```

1 <RootW>:
2     BoxLayout:
3         orientation: 'vertical'
4         Button:
5             id: bt1
6             text: 'Button1'
7             on_release: root.funcButton(tx2)
8         TextInput:
9             id: tx1
10            on_text:      root.funcTextInput(tx2,tx1.text)
11        TextInput:
12            id: tx2

```

このように、インスタンス名、プロパティ、イベントなどを含め、GUI の階層構造が簡潔に記述できる。この Kv ファイルを読み込んで GUI を実装するアプリケーション (Python プログラム) を kivyKV01.py に示す。

プログラム: kivyKV01.py

```

1 # coding: utf-8
2 #----- 関連モジュールの読み込み -----
3 from kivy.app import App
4 from kivy.uix.boxlayout import BoxLayout
5 from kivy.core.window import Window
6
7 #----- GUIの構築 -----
8 class RootW(BoxLayout):
9     # コールバック関数
10    def funcButton(self,tx2):
11        tx2.text = 'result: Button1 was touched'
12    def funcTextInput(self,tx2,t):

```

```

13         tx2.text = 'result: ' + t
14
15 #----- アプリケーション本体 -----
16 class KivyKV01App(App):
17     def build(self):
18         return RootW()
19
20 # アプリの実行
21 Window.size = (300,100)
22 KivyKV01App().run()

```

解説：

Kv ファイルの 1,2 行目が、Python プログラムの 8 行目のクラス定義に対応している。Python プログラムでは、GUI の最上位オブジェクトのクラス宣言と、コールバック関数の定義のみを記述しており、GUI の階層構造は全て Kv ファイル内に記述している。

■ Kv ファイルにおけるクラスの定義

‘<...>’ の記述は Kv における規則の記述である。この記述を応用することで、Python プログラム側のクラス定義に対応させることができる。

■ Kv ファイルにおける id

Kv ファイル内では、ウィジェットなどに識別名を与えるために ‘id:’ を記述する。(id により与えられた識別名は、厳密にはインスタンス名ではない)

■ Kv ファイルにおけるイベントハンドリング

‘イベント名:’ に続いて呼び出すコールバック関数を記述する。このとき、Python プログラムの中のどこに記述された関数（メソッド）かを明示するために root という指定をしている。この root は Kv において記述対象のツリーの最上位を意味するものであり、Python 側プログラムでは、これを用いたクラス RootW（Python 側の 9 行目）が対応する。すなわち、root.funcButton(tx2, bt1.text) は、RootW クラスのメソッド funcButton を呼び出すことになる。

Kv 側からの関数（メソッド）呼び出しにおいては自由に引数を与えることができ、それを受ける Python 側の関数（メソッド）の仮引数も対応する形に記述する。ただし Python 側の仮引数には、第 1 引数として、それを呼び出したオブジェクト自身（self）を受け取る仮引数を記述する必要がある、結果として、引数の数が 1 つ多い記述となる。

3.7.1.2 Python プログラムと Kv ファイルの対応

先の例では、アプリケーションのクラス名が KivyKV01App なので、それに対応する Kv ファイルの名前は KivyKV01.kv とする。すなわち、アプリケーションのクラスの名前は、

‘任意の名前 App’

と末尾に ‘App’ を付ける。そして対応する Kv ファイルの名前には、App の前の部分に拡張子 ‘.kv’ を付けたものとする。こうすることで、アプリケーションの起動時に自動的に対応が取られて Kv ファイルが読み込まれる。

Builder クラスを用いた Kv ファイルの読み込み

Builder クラスの load_file メソッドを使用することで、先に説明した名前の制限にとらわれることなく、Python 側、Kv 側共に自由にファイル名を付けることができる。Builder クラスを使用するためには次のようにして必要なモジュールを読み込んでおく。

```
from kivy.lang.builder import Builder
```

この後、Python 側プログラムの冒頭で、

```
Builder.load_file( 'Kv ファイル名' )
```

とすることで、指定したファイルから Kv の記述を読み込むことができる。

3.8 時間によるイベント

Kivy には Clock モジュールがあり、ユーザからの入力以外に時間によるイベントハンドリングが可能である。すなわち、設定された時間が経過したことをイベントとしてハンドリングすることができ、いわゆる**タイマー**の動作を実現することができる。Clock モジュールを使用するには次のようにして必要なものを読み込んでおく。

```
from kivy.clock import Clock
```

3.8.1 時間イベントのスケジュール

ClockEvent オブジェクトとして時間イベントを生成することで、指定した時間が経過した時点でコールバック関数を起動することができる。

《 コールバック関数のスケジュール 》

一度だけ： `Clock.schedule_once (コールバック関数, 経過時間)`

繰り返し： `Clock.schedule_interval (コールバック関数, 経過時間)`

この結果、ClockEvent オブジェクトが返される。経過時間の単位は「秒」であり、浮動小数点数で表現する。コールバック関数は仮引数を 1 つ取る形で定義しておく。コールバック関数には起動時に経過時間が引数として渡される。コールバック関数は戻り値として `True` を返すように定義するが、`False` を返す形にすると `schedule_interval` によってスケジュールされた時間イベントがキャンセルされる。

`schedule_interval` によってコールバック関数の繰り返し起動がスケジュールされた場合、得られた ClockEvent オブジェクトに対して `cancel` メソッドを使用することで、スケジュールを解除（キャンセル）することができる。また、

`Clock.unschedule(スケジュールされた ClockEvent)`

とすることでもキャンセルできる。

時間イベントの実装例は「3.9.4 スワイプ」(p.229) のところで紹介する。

3.9 GUI 構築の形式

Kivy は通常の PC だけでなく、スマートフォンやタブレット PC といった携帯情報端末のためのアプリケーション開発を視野に入れているため、独特の UI デザインを提供する。例えば、複数のウィンドウを同時に表示する形式ではなく、1 つのウィンドウ内で、UI をまとめた**スクリーン**を切り替える形式などが特徴的である。

ここでは、実用的な UI インターフェースを構築するためのいくつかの形式について説明する。

3.9.1 スクリーンの扱い： Screen と ScreenManager

Kivy では、UI の 1 つのまとまりを Screen として扱い、それらをスライドのようにして切り替えることが可能である。Screen オブジェクトには各種のレイアウトオブジェクトを登録することができ、各々のレイアウトにはこれまで説明した方法で UI を構築する。複数作成した Screen オブジェクトは 1 つの ScreenManager オブジェクトに登録して管理する。ScreenManager に登録されたスクリーンは transition によって切り替えることができる。

ScreenManager, Screen 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んでおく。

```
from kivy.uix.screenmanager import ScreenManager, Screen
```

3.9.1.1 ScreenManager

ScreenManager オブジェクトは複数の Screen を登録管理するもので、次のようにして生成する。

```
sm = ScreenManager()
```

この例では、生成された ScreenManager オブジェクトが `sm` に保持されている。

3.9.1.2 Screen

Screen オブジェクトは、先の ScreenManager に登録して使用する。このオブジェクトが1つの UI スクリーンとなり、更にここにレイアウトオブジェクトなどを登録する。Screen オブジェクトは次のようにして生成する。

```
sc1 = Screen(name=識別名)
```

「識別名」は Screen を識別するためのもので文字列として与える。生成した Screen オブジェクトは add_widget メソッドで ScreenManager オブジェクトに登録する。

ScreenManager オブジェクトのプロパティ current に Screen の識別名を与えることで表示する Screen を選択できる。また、transition プロパティの設定により、Screen が切り替わる様子を制御できる。

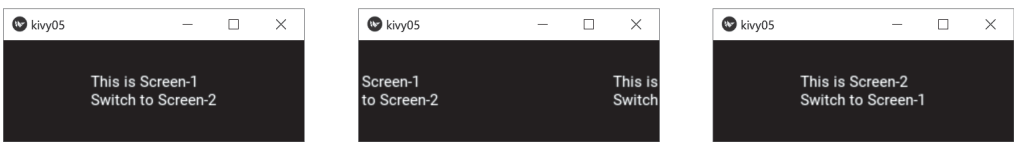
Screen の遷移の効果は、いわゆるスライドインであるが、その他の効果も設定できる。

参考) 遷移の効果

Kivy の ScreenManager には表 30 のような様々な transition が用意されている。これらは ScreenManager オブジェクト生成時に設定するが、詳しくは巻末付録に挙げている Kivy のサイトを参照のこと。

表 30: 遷移効果の効果	
遷移	効果
SlideTransition	縦／横方向のスライディング（デフォルト）
SwapTransition	iOS のスワップに似た切り替え
FadeTransition	フェードイン／アウトによる切り替え
WipeTransition	ワイプによる切り替え
FallOutTransition	消え行くような切り替え
RiselnTransition	透明から不透明に変化するような切り替え
NoTransition	瞬時の切り替え
CardTransition	上に重ねるような切り替え

次に、図 32 のように、2つのスクリーンを切り替え表示するプログラム kivy05-1.py を挙げて Screen の扱いについて説明する。



スクリーンをタッチすると、 → スライディングが起こり、 → 次のスクリーンに切り替わる

図 32: スクリーンを切り替えるアプリケーションの実行例

プログラム：kivy05-1.py

```
1 # coding: utf-8
2 #----- 必要なパッケージの読み込み -----
3 from kivy.app import App
4 from kivy.uix.screenmanager import ScreenManager, Screen
5 from kivy.uix.anchorlayout import AnchorLayout
6 from kivy.uix.label import Label
7 from kivy.core.window import Window
8
9 #----- アプリケーションクラスの定義 -----
10 class kivy05(App):
11     def build(self):
12         return root
13
14 #----- GUIの構築 -----
15 # ウィンドウのサイズ
16 Window.size = (300,100)
17
```



```

18 # スクリーンマネージャ
19 root = ScreenManager()
20 # スクリーン
21 sc1 = Screen(name='screen_1')
22 sc2 = Screen(name='screen_2')
23 # レイアウト
24 al1 = AnchorLayout()
25 al2 = AnchorLayout()
26 # ラベル
27 l1 = Label(text='This is Screen-1\nSwitch to Screen-2')
28 l2 = Label(text='This is Screen-2\nSwitch to Screen-1')
29 # 組み立て
30 al1.add_widget(l1)
31 al2.add_widget(l2)
32 sc1.add_widget(al1)
33 sc2.add_widget(al2)
34 root.add_widget(sc1)
35 root.add_widget(sc2)
36 # 最初に表示されるスクリーン
37 root.current = 'screen_1'
38
39 #----- スクリーン遷移の処理 -----
40 # コールバック関数
41 def callbk1(self,t):      # screen-1からscreen-2へ
42     root.transition.direction = 'left'
43     root.transition.duration = 3      # ゆっくり
44     root.current = 'screen_2'
45 def callbk2(self,t):      # screen-2からscreen-1へ
46     root.transition.direction = 'right'
47     root.transition.duration = 0.4   # デフォルト
48     root.current = 'screen_1'
49 # ラベルオブジェクトに登録
50 l1.bind(on_touch_up=callbk1)
51 l2.bind(on_touch_up=callbk2)
52
53 #----- アプリケーションの実行 -----
54 ap = kivy05()
55 ap.run()

```

解説

19～35行目でスクリーンとUIを構築している。41～48行目でスクリーンを切り替えるためのコールバック関数を定義して、Labelオブジェクトに登録している。この例でわかるように、ScreenManagerのプロパティ transition.direction で遷移の方向を、transition.duration で遷移にかかる時間（秒）を設定する。

ScreenManager とは別に、より簡単にスワイプを実現する方法を「3.9.4 スワイプ」（p.229）のところで解説する。

3.9.2 アクションバー： ActionBar

一般的な GUI アプリケーションで採用されているプルダウンメニューに近い機能を Kivy ではアクションバー (ActionBar) という形で実現する。ActionBar を構築するために必要なクラスは次に挙げる 5 つのものである。

ActionBar, ActionView, ActionPrevious, ActionGroup, ActionButton

これらのクラスを使用するために、必要なモジュールを次のようにして読み込む。

```

from kivy.uix.actionbar import ActionBar, ActionView, ActionPrevious, ¥
                                ActionGroup, ActionButton

```

ここでは、よく知られた GUI における「メニューバー」「メニュー」「メニュー項目」の構成に対応させる形で ActionBar の構築について説明する。

一般的に「メニューバー」と呼ばれるものは、例えば「ファイル」「編集」…といった「メニュー」を配置しており、それら各メニューをクリックするとプルダウンメニューが表示されて、その中に「新規」「開く」「保存」「閉じる」…といった「メニュー項目」が並んでいる。この場合の「メニューバー」は Kivy の ActionBar に相当する。ActionBar にはまず ActionView を登録して、それに対して ActionPrevious, ActionGroup を登録する。この ActionGroup が一般的な「メニュー」に相当する。その後 ActionGroup に対していわゆる「メニュー項目」に相当する ActionButton を

必要なだけ登録する。

ActionBar を構築する様子を階層的に示すと次のようになる。

【ActionBar の構築】

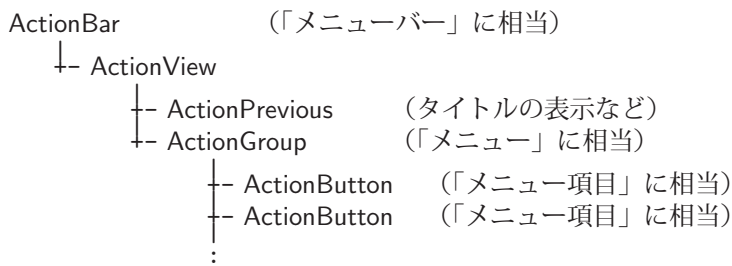
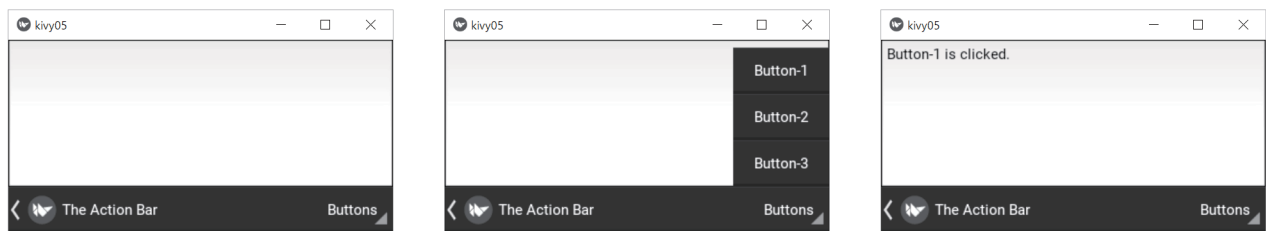


図 33 のような ActionBar を実装するアプリケーションを例に挙げて構築方法について説明する。



ButtonGroup (右下) をクリックすると、 → ActionButton が表示される。 → 選択によるコールバック処理

図 33: ActionBar の実装例

これを実装したプログラムを kivy05-2.py に示す。

プログラム：kivy05-2.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  from kivy.app import App
4  from kivy.uix.actionbar import ActionBar, ActionView, ActionPrevious, \
5      ActionGroup, ActionButton
6  from kivy.uix.boxlayout import BoxLayout
7  from kivy.uix.anchorlayout import AnchorLayout
8  from kivy.uix.textinput import TextInput
9  from kivy.core.window import Window
10
11 #----- アプリケーションクラスの定義 -----
12 class kivy05(App):
13     def build(self):
14         return root
15
16 #----- GUIの構築 -----
17 # ウィンドウのサイズ
18 Window.size = (400,200)
19
20 # 最上位のレイアウト
21 root = BoxLayout(orientation='vertical')
22
23 # テキストフィールド
24 txt = TextInput()
25
26 # アクションバー
27 acBar = ActionBar()
28 acView = ActionView()
29 acPrev = ActionPrevious(title='The Action Bar')
30 acGrp = ActionGroup(text='Buttons', mode='spinner')
31 acBtn1 = ActionButton(text='Button-1')
32 acBtn2 = ActionButton(text='Button-2')
33 acBtn3 = ActionButton(text='Button-3')
34
35 # 組み立て
36 root.add_widget(txt)
37 acGrp.add_widget(acBtn1)
```

```

38 acGrp.add_widget(acBtn2)
39 acGrp.add_widget(acBtn3)
40 acView.add_widget(acPrev)
41 acView.add_widget(acGrp)
42 acBar.add_widget(acView)
43 root.add_widget(acBar)
44
45 # コールバック関数
46 def clback1(self):
47     txt.text = 'Button-1 is clicked.'
48 def clback2(self):
49     txt.text = 'Button-2 is clicked.'
50 def clback3(self):
51     txt.text = 'Button-3 is clicked.'
52
53 acBtn1.bind(on_release=clback1)
54 acBtn2.bind(on_release=clback2)
55 acBtn3.bind(on_release=clback3)
56
57 #----- アプリケーションの実行 -----
58 ap = kivy05()
59 ap.run()

```

解説

27～33 行目で ActionBar 構築に必要なオブジェクトを生成している。ActionGroup（いわゆるメニュー）を生成する際に、キーワード引数 'text=' を与えることでグループ名を設定することができ、これが ActionBar 上に表示される。また、キーワード引数 mode='spinner' を与えると、ActionGroup クリック時に ActionButton（いわゆるメニュー項目）が「立ち上がるメニュー」として表示される。

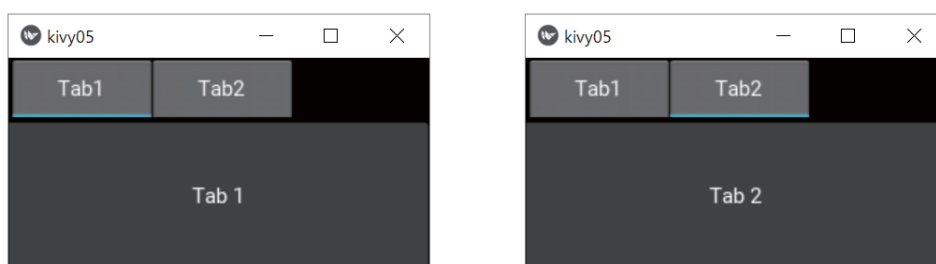
3.9.3 タブパネル： TabbedPanel

TabbedPanel を使用すると、いわゆる切り替えタブが実現できる。TabbedPanel には TabbedPanelItem オブジェクトを必要な数だけ登録する。各 TabbedPanelItem オブジェクトが個々のタブパネルであり、これにレイアウトオブジェクトを配置することができる。

タブパネルを構築するには、次のようにして必要なモジュールを読み込んでおく。

```
from kivy.uix.tabbedpanel import TabbedPanel, TabbedPanelItem
```

ここでは、図 34 のように、2 つのタブを切り替えるプログラムを例に挙げる。



タブの切り替え表示

図 34: TabbedPanel の実装例

実装したプログラムを kivy05-3.py に示す。

プログラム：kivy05-3.py

```

1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  from kivy.app import App
4  from kivy.uix.tabbedpanel import TabbedPanel, TabbedPanelItem
5  from kivy.uix.anchorlayout import AnchorLayout
6  from kivy.uix.label import Label
7  from kivy.core.window import Window
8
9  #----- アプリケーションクラスの定義 -----

```

```

10 class kivy05(App):
11     def build(self):
12         return root
13
14 # タブ パネル
15 root = TabbedPanel()
16 root.do_default_tab = False
17 # タブ1
18 ti1 = TabbedPanelItem(text='Tab1')
19 lb1 = Label(text='Tab 1')
20 ti1.add_widget(lb1)
21 root.add_widget(ti1)
22 # タブ2
23 ti2 = TabbedPanelItem(text='Tab2')
24 lb2 = Label(text='Tab 2')
25 ti2.add_widget(lb2)
26 root.add_widget(ti2)
27
28 root.default_tab = ti1
29
30 #----- GUIの構築 -----
31 # ウィンドウのサイズ
32 Window.size = (300,150)
33 ap = kivy05()
34 ap.run()

```

解説

18,23 行目のように、TabbedPanelItem のオブジェクト生成時にキーワード引数 'text=' を与えることで、タブの見出しを設定できる。

3.9.4 スワイプ： Carousel

Carousel による UI は ScreenManager による UI と似ているが、スワイプ機能が予め備わっており、ScreenManager の transition を使用するよりも UI の実装が容易であることが特徴である。

Carousel オブジェクトにはレイアウトをはじめとするウィジェットを登録するが、それらは登録した順に順序付けされる。そして、Carousel オブジェクトに対して、load_next、load_previous といったメソッドを使用することで、登録されたウィジェット群を順番に（逆順に）「回転」させる¹⁷⁶ ことができる。

Carousel を使用したプログラム kivy05-4.py を次に示す。

プログラム：kivy05-4.py

```

1  # coding: utf-8
2  #----- 必要なパッケージの読み込み -----
3  from kivy.app import App
4  from kivy.uix.carousel import Carousel
5  from kivy.uix.label import Label
6  from kivy.clock import Clock
7  from kivy.core.window import Window
8
9  #----- アプリケーションクラスの定義 -----
10 class kivy05(App):
11     def build(self):
12         return root
13
14 # Carouselの生成
15 root = Carousel(direction='right')
16 lb1 = Label(text='Slide - 1')
17 lb2 = Label(text='Slide - 2')
18 lb3 = Label(text='Slide - 3')
19 root.add_widget(lb1)
20 root.add_widget(lb2)
21 root.add_widget(lb3)
22 root.loop = True      # 循環の切り替え指定
23

```

¹⁷⁶”carousel”（英）は「回転木馬」という意味である。

```

24 #----- 自動スワイプ -----
25 # コールバック関数（引数にはdurationが与えられる）
26 def autoSwipe(dt):
27     print(dt)
28     root.load_next()
29     return True      # Falseを返すとスケジュールがキャンセルされる
30 # スケジュール
31 tm = Clock.schedule_interval(autoSwipe, 3)
32 print(type(tm))      # 型の確認
33
34 # 停止用コールバック関数
35 def cancelSwipe(self,t):
36     print('自動スワイプがキャンセルされました. ')
37     Clock.unschedule(tm)      # キャンセル方法1
38     # tm.cancel()              # キャンセル方法2
39 root.bind(on_touch_down=cancelSwipe)
40
41 #----- GUIの構築 -----
42 # ウィンドウのサイズ
43 Window.size = (300,150)
44 ap = kivy05()
45 ap.run()

```

解説

22行目にあるように、Carousel オブジェクトの `loop` プロパティに `True` を設定すると、末尾のウィジェットの次は先頭のウィジェットに戻るように順序付けされ、ウィジェット群を順番に回転させることができる。

4 実用的なアプリケーション開発に必要な事柄

4.1 日付と時間に関する処理

Python では日付や時刻、あるいは経過時間といった情報を扱うことができる。ここではそのためのモジュールである `datetime` モジュール、`time` モジュール、`timeit` モジュールについて解説する。

4.1.1 日付と時刻の取り扱い： `datetime` モジュール

日付と時刻の情報を扱うために `datetime` モジュールが標準的に提供されている。このモジュールは様々なクラスや関数を提供しており、

```
from datetime import *
```

のように実行して当該モジュール内の全てのクラスや関数を読み込むことができるが、他のライブラリと併用する場合に**名前の衝突**¹⁷⁷ が起こることがあることに注意しなければならない。このモジュールが提供するクラスの内、特に `datetime` クラス、`timedelta` クラス、`timezone` クラスを用いることが多く、次のようにしてこれらクラスを選択的に読み込むと安全である。

例. モジュールの読み込み

```
>>> from datetime import datetime, timedelta, timezone Enter
```

以後の解説では、これらモジュールを読み込んだ状態を前提とする、

`datetime` モジュールで扱える情報は年、月、日、時、分、秒、マイクロ秒の7つの要素である。例えば現在時刻を取得するには `now` メソッドを使用する。

例. 現在時刻の取得

```
>>> datetime.now() Enter ←現在時刻の取得  
datetime.datetime(2017, 5, 5, 13, 10, 25, 653093) ←処理結果
```

この例のように時間情報は `datetime.datetime(...)` という形式で（`datetime` クラスのオブジェクトとして）扱われる。

4.1.1.1 `datetime` オブジェクトの分解と合成

`datetime` オブジェクトを「日付のみ」のオブジェクト（`date` オブジェクト）と「日付なし時刻」のオブジェクト（`time` オブジェクト）に分解することができる。

例. 時刻情報の分解（先の例の続き）

```
>>> d = datetime.now() Enter ←現在時刻の取得  
>>> dt = d.date() Enter ←日付のみの取り出し  
>>> dt Enter ←内容確認  
datetime.date(2017, 5, 5) ←結果  
>>> tm = d.time() Enter ←日付なし時刻の取り出し  
>>> tm Enter ←内容確認  
datetime.time(13, 23, 16, 396945) ←結果
```

このように、`datetime` オブジェクトに対して `date` メソッドや `time` メソッドを実行する。その結果、それぞれ

```
datetime.date(...), datetime.time(...)
```

という形式（それぞれ `date` クラス、`time` クラスのオブジェクト）で結果が得られる。またこれらを `combine` メソッドで連結して `datetime` オブジェクトを作成することもできる。

例. `date` オブジェクトと `time` オブジェクトの連結（先の例の続き）

```
>>> datetime.combine(dt,tm) Enter ←日付と時刻を連結  
datetime.datetime(2017, 5, 5, 13, 23, 16, 396945) ←連結結果
```

`now` メソッドで現在時刻を取得することとは別に、指定した特定の日付、時間を構成することもできる。

¹⁷⁷ 詳しくは巻末付録「D.1 ライブラリの読み込みに関すること」を参照のこと。

例. 「1966 年 3 月 14 日」という日付と時刻の生成

```
>>> datetime(1966,3,14) Enter ←日付情報の生成
datetime.datetime(1966, 3, 14, 0, 0) ←得られたデータ
```

4.1.1.2 文字列を datetime オブジェクトに変換する方法

strptime メソッドを使用すると、日付と時刻の情報を含んだ文字列から datetime オブジェクトを作成することができる。

書き方: `strptime(日付時刻を含んだ文字列, 書式文字列)`

この関数は、第 1 引数に与えた「日付時刻を含んだ文字列」から第 2 引数「書式文字列」に沿った形で日付時刻を取り出し、結果を datetime オブジェクトとして返す。この関数の実行例を次に示す。

例. strptime 関数

```
>>> datetime.strptime('2022-03-14 13:15:10', '%Y-%m-%d %H:%M:%S') Enter
datetime.datetime(2022, 3, 14, 13, 15, 10) ←得られたデータ
```

この例のように、書式文字列には「%」で始まる**書式文字列**を記述し、文字列中の各情報の場所を指定する。書式文字列として重要なものを表 31 (p.233) に示す。

■ ISO8601 形式の日付、時刻の表現

日付と時刻の表現に関する国際規格に ISO8601 がある。これは

日付 T 時刻

の形式である。例えば「2022 年 3 月 14 日 13 時 15 分 12 秒」は次のように表現される。

2022-03-14T13:15:12

Python 3.7 から datetime クラスに fromisoformat メソッドが導入され、この形式で表現された文字列から datetime オブジェクトを作成することができる。

例. ISO8601 形式から datetime オブジェクトを作成する

```
>>> datetime.fromisoformat('2022-03-14T13:15:12') Enter ← ISO8601 フォーマットを変換
datetime.datetime(2022, 3, 14, 13, 15, 12) ←変換結果
```

4.1.1.3 datetime オブジェクトの差: timedelta

マイナス演算子「-」で 2 つの datetime オブジェクトの差を求めることで、2 つの時刻の間の時間差を取得することができる。

例. 1966 年 3 月 14 日 13 時 15 分から 2017 年 5 月 5 日 14 時 00 分までの経過時間

```
>>> d1 = datetime(1966,3,14,13,15) Enter ←日付情報 1966/03/14 13:15 の生成
>>> d2 = datetime(2017,5,5,14,0) Enter ←日付情報 2017/05/05 14:00 の生成
>>> td = d2 - d1 Enter ←経過時間の取得
>>> td Enter ←値の確認
datetime.timedelta(days=18680, seconds=2700) ←経過日数が「18680 日と 2700 秒」である
```

このように、自然な形の減算で時間差を取得することができ、結果は

`datetime.timedelta(days=日数, seconds=秒数)`

という形式 (timedelta クラスのオブジェクト) で得られ、days, seconds, microseconds といった属性を持つ。

例. timedelta オブジェクトの属性 (先の例の続き)

```
>>> td.days Enter ←日数
18680 ← 18680 日
>>> td.seconds Enter ←秒
2700 ← 2700 秒
>>> td.microseconds Enter ←マイクロ秒
0 ←秒の小数点以下は 0
```

注意) timedelta の days, seconds, microseconds 属性は事後で変更はできない。(readonly attribute である)

timedelta を用いると「～日後の日付」や「～日前の日付」を算出することもできる。計算方法は、datetime オブジェクトに対する timedelta オブジェクトの加算あるいは減算である。

例. 1966 年 3 月 14 日 13 時 15 分から 18680 日と 2700 秒経過した日付と時刻

```
>>> d1 = datetime(1966,3,14,13,15)  ←日付情報 1966/03/14 13:15 の生成
>>> td = timedelta(18680, 2700)      ←経過した日数と秒数の生成
>>> d1 + td                           ←日付の取得
datetime.datetime(2017, 5, 5, 14, 0)  ←得られた日付
```

この例のように、timedelta(日数, 秒数) の形¹⁷⁸ で時間差を作成することもできる。

4.1.1.4 日付, 時刻の書式整形

日付, 時刻を書式整形して文字列で取得することができる。

例. 日付, 時刻の書式整形

```
>>> d1 = datetime(1966,3,14,13,15)  ←日付情報の生成
>>> str(d1)                          ←文字列に変換
'1966-03-14 13:15:00'                ←得られた文字列
>>> d1.ctime()                      ←文字列 (UNIX 形式, C 言語) に変換
'Mon Mar 14 13:15:00 1966'          ←得られた文字列
```

このように str 関数の引数に datetime オブジェクトを与える。あるいは C 言語 (UNIX) の形式で書式整形するには ctime メソッドを用いる。更に strftime メソッドを使用すると、より自由な書式整形ができる。

例. strftime による書式整形

```
>>> d2 = datetime.now()              ←現在時刻の取得
>>> d2.strftime('%Y %m %d %a %H %M %S %f') ←書式整形
'2018 05 26 Sat 12 57 46 243601'     ←整形結果の文字列
```

このように ‘%’ で記述した書式指定を strftime の引数に与えることによって書式整形ができる。書式指定の一部を表 31 に示す。(曜日や月名はロケール¹⁷⁹ の設定の影響を受ける)

表 31: 日付, 時刻の書式指定

書式	意味	書式	意味	書式	意味
%Y	年	%m	月	%d	日
%H	時	%M	分	%S	秒
%a	曜日 (短縮)	%A	曜日	%f	マイクロ秒
%b	月名 (短縮)	%B	月名		

Python 3.7 から datetime クラスに isoformat メソッドが導入され、datetime の値を ISO8601 形式の文字列に変換することができる。

例. datetime オブジェクトを ISO8601 形式に変換する

```
>>> d = datetime.now()              ←現在時刻を取得
>>> d.isoformat()                   ←ISO8601 形式に変換
'2022-03-10T22:58:49.280558'       ←変換結果
```

4.1.1.5 datetime のプロパティ

datetime オブジェクトから各要素 (年, 月, 日, 時, 分, 秒, マイクロ秒) を取り出すための各種のプロパティがある。

¹⁷⁸timedelta のコンストラクタには第 3 引数としてマイクロ秒の値を与えることもできる。

¹⁷⁹詳しくは p.244 「4.2 ロケール (locale)」参照のこと。

例. datetime からの要素の取り出し

```
>>> d = datetime.now() [Enter]    ←現在の日付, 時刻の取得
>>> d [Enter]    ←確認
datetime.datetime(2017, 5, 5, 14, 54, 23, 305326)    ←表示結果

>>> d.year [Enter]    ←「年」の取得
2017    ←表示結果 (年)
>>> d.month [Enter]    ←「月」の取得
5    ←表示結果 (月)
>>> d.day [Enter]    ←「日」の取得
5    ←表示結果 (日)
>>> d.hour [Enter]    ←「時」の取得
14    ←表示結果 (時)
```

```
>>> d.minute [Enter]    ←「分」の取得
54    ←表示結果 (分)
>>> d.second [Enter]    ←「秒」の取得
23    ←表示結果 (秒)
>>> d.microsecond [Enter]    ←「マイクロ秒」の取得
305326    ←表示結果 (マイクロ秒)
```

4.1.1.6 タイムゾーンについて

世界各地の日付と時刻は協定世界時 (UTC)¹⁸⁰ を基準にしている。datetime クラスの now メソッドは使用している計算機のローカルタイムを取得するものであり、日本国内の計算機でこれを実行すると、UTC の時刻から 9 時間進んだもの (日本時間) が得られる。UTC の現在時刻を取得するには utcnow メソッドを使用する。次に示す例は日本国内の計算機環境で utcnow, now メソッドを実行したものである。

例. UTC とローカルタイムの取得

```
>>> d1 = datetime.utcnow(); d2 = datetime.now() [Enter]    ← UTC と日本時間を同時に取得
>>> d1 [Enter]    ← UTC の日付時刻を確認
datetime.datetime(2022, 3, 10, 12, 18, 33, 645845)    ← UTC の日付時刻
>>> d2 [Enter]    ←日本時間の日付時刻を確認
datetime.datetime(2022, 3, 10, 21, 18, 33, 645845)    ←日本時間の日付時刻
```

日本時間の方が UTC よりも 9 時間進んでいることがわかる。

■ datetime オブジェクトのタイムゾーンの属性

datetime オブジェクトにはタイムゾーンを表す tzinfo 属性がある。この属性の値は timezone クラスのオブジェクトを保持する。timezone クラスのオブジェクトはローカルタイムを表現するもので、生成する際に UTC との時間差を timedelta オブジェクトとして与える。timezone オブジェクトを単体で作成する例を次に示す。

例. timezone オブジェクトの作成

```
>>> from datetime import datetime, timedelta, timezone [Enter]    ←必要なクラスの読み込み
>>> z = timezone( timedelta(hours=9) ) [Enter]    ← 9 時間の時間差のタイムゾーンを作成
>>> z [Enter]    ←内容確認
datetime.timezone(datetime.timedelta(seconds=32400))    ←得られたタイムゾーン情報
```

この例では UTC からの時間差が 9 時間であるタイムゾーンの情報を変数 z に得ている。次に、timezone オブジェクトをタイムゾーンの属性として持つ datetime オブジェクトを作成する例を示す。

例. タイムゾーン属性を持つ現在時刻の取得 (先の例の続き)

```
>>> d = datetime.now( tz=z ) [Enter]    ←タイムゾーン情報を与えて現在時刻を取得
>>> d [Enter]    ←内容確認
datetime.datetime(2022, 3, 11, 12, 37, 13, 250186,    ←得られた datetime オブジェクト
tzinfo=datetime.timezone(datetime.timedelta(seconds=32400)))
```

この例のように now メソッドの引数に「tz=タイムゾーン情報」を与えることで、得られる datetime オブジェクトにタイムゾーンの属性 (tzinfo) を与えることができる。

tzinfo 属性は datetime オブジェクトを作成する段階で与えるものであり、事後に与えることはできない。(次の例)

¹⁸⁰ グリニッジ標準時 (GMT) を置き換えるものとして使用されている。

例. tzinfo 属性を事後に与える試み（先の例の続き）

```
>>> d = datetime.now() [Enter] ←タイムゾーン情報を与えずに現在時刻を取得
>>> d.tzinfo = z [Enter] ←tzinfo 属性を設定しようとする…

Traceback (most recent call last): ←エラーとなる
  File "<stdin>", line 1, in <module>
AttributeError: attribute 'tzinfo' of 'datetime.datetime' objects is not writable
```

このように、既存の datetime オブジェクトの tzinfo 属性は変更できない旨のエラーが発生する。また、タイムゾーンを指定せずに作成した datetime オブジェクトの tzinfo 属性は None となる。（次の例）

例. タイムゾーン属性を持たない datetime オブジェクトの tzinfo 属性（先の例の続き）

```
>>> print( d.tzinfo ) [Enter] ←tzinfo 属性の確認
None ←参照結果
```

datetime クラスのコンストラクタに引数「tzinfo=タイムゾーン情報」を与えることで tzinfo 属性を与えることができる。（次の例）

例. コンストラクタにタイムゾーンを与える（先の例の続き）

```
>>> datetime(2022,3,14,13,15,12, tzinfo=z ) [Enter]
datetime.datetime(2022, 3, 14, 13, 15, 12, ←得られた datetime オブジェクト
                tzinfo=datetime.timezone(datetime.timedelta(seconds=32400)))
```

ISO8601 形式はタイムゾーンを扱うことができる。

表記： 日付 T 時刻 ± UTC からのオフセット

この表記による文字列を fromisoformat メソッドで datetime に変換する例を次に示す。

例. タイムゾーン指定のある ISO8601 表記から datetime オブジェクトを生成する

```
>>> datetime.fromisoformat('2022-03-14T13:15:12+09:00') [Enter] ←タイムゾーン付き ISO8601 表記
datetime.datetime(2022, 3, 14, 13, 15, 12, ←得られた datetime オブジェクト
                tzinfo=datetime.timezone(datetime.timedelta(seconds=32400)))
```

■ タイムゾーンの変換

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

例. 日本時間の取得

```
>>> djpn = datetime.now( tz=datetime.timezone(datetime.timedelta(hours=9)) ) [Enter] ←現在時刻の取得（日本時間）
>>> djpn [Enter] ←確認
datetime.datetime(2022, 3, 11, 14, 58, 2, 874671, ←得られた datetime オブジェクト
                tzinfo=datetime.timezone(datetime.timedelta(seconds=32400)))
```

この例では日本時間の datetime オブジェクトが変数 djpn に得られている。これを UTC に変換する例を次に示す。

例. UTC への変換（先の例の続き）

```
>>> dutc = djpn.astimezone( tz=datetime.timezone(datetime.timedelta(hours=0)) ) [Enter] ← UTC に変換
>>> dutc [Enter] ←確認
datetime.datetime(2022, 3, 11, 5, 58, 2, 874671, tzinfo=datetime.timezone.utc) ← UTC 時刻
```

先に得られた日本時間よりも 9 時間遅い時刻（UTC）が変数 dutc に得られている。また、UTC のタイムゾーンは実行例にあるように「datetime.timezone.utc」と表されている。これは `timezone(datetime.timedelta(hours=0))` と同じものを意味する。

■ zoneinfo モジュール

Python3.9 からタイムゾーンを扱うための zoneinfo モジュールが標準添付¹⁸¹されており、タイムゾーン ID¹⁸² が使用できる。具体的には、zoneinfo モジュールの ZoneInfo クラスのオブジェクトとしてタイムゾーン情報を扱う。（次

¹⁸¹Windows 環境でこれを使用する場合は tzdata モジュールをインストールしておく必要がある。

PIP コマンドを使用する場合は「`pip install tzdata`」などとしてインストールする。

¹⁸²タイムゾーンを地域、国などの名前で表現した文字列。IANA が規定している。

の例)

例. 日本時間のタイムゾーンを作成する

```
>>> from zoneinfo import ZoneInfo Enter    ← ZoneInfo クラスの読み込み
>>> z = ZoneInfo('Asia/Tokyo') Enter    ←タイムゾーン ID 'Asia/Tokyo' のタイムゾーンを作成
>>> z Enter    ←確認
zoneinfo.ZoneInfo(key='Asia/Tokyo')    ←得られたタイムゾーン
```

この例における 'Asia/Tokyo' が日本時間のタイムゾーン ID であり、これを元にしてタイムゾーンを変数 *z* に作成している。次に、これを与えて *datetime* オブジェクトを作成する例を示す。

例. *ZoneInfo* オブジェクトを使用して *datetime* オブジェクトを作成する (先の例の続き)

```
>>> from datetime import datetime, timedelta, timezone Enter    ←必要なクラスの読み込み
>>> djpn = datetime.now( tz=z ) Enter    ←タイムゾーンを与えて現在時刻を取得
>>> djpn Enter    ←確認
datetime.datetime(2022, 3, 11, 16, 10, 7, 276301,          ←得られた datetime オブジェクト
                  tzinfo=zoneinfo.ZoneInfo(key='Asia/Tokyo'))
```

これを UTC 時刻に変換して内容を確認する。

例. 先の例の時刻を UTC に変換する (先の例の続き)

```
>>> dutc = djpn.astimezone( tz=timezone.utc ) Enter    ← UTC に変換
>>> dutc Enter    ←確認
datetime.datetime(2022, 3, 11, 7, 10, 7, 276301, tzinfo=datetime.timezone.utc)
```

先に作成した *djpn* よりも *dutc* の方が 9 時間遅れており、*djpn* が日本時間を正しく表現していることがわかる。

zoneinfo モジュールは、使用可能な全てのタイムゾーン ID をセットの形で取得する *available_timezones* 関数を提供する。

例. 使用可能な全てのタイムゾーン ID を出力する

```
>>> from zoneinfo import available_timezones Enter    ←関数 available_timezones の読み込み
>>> available_timezones() Enter    ←実行
{'Asia/Yangon', 'America/Denver', 'America/Matamoros', 'Atlantic/Azores',
 'Australia/Queensland', 'America/Indiana/Petersburg', 'America/Campo_Grande',
  ⋮
 (途中省略)
  ⋮
 'America/Indiana/Marengo', 'Africa/Dar_es_Salaam', 'Etc/GMT+9', 'Asia/Baku',
 'America/Bahia_Banderas', 'Africa/Bangui', 'Eire', 'GB-Eire', 'Europe/Zagreb'}
```

課題. 関数 *available_timezones()* が返すセットを元にして、タイムゾーン ID 毎に UTC との時差を出力する処理を実装せよ。

■ 処理系のローカルタイムゾーンの取得

Lennart Regebro 氏が開発した *tzlocal* モジュール¹⁸³ を使用することで処理系のローカルタイムゾーンを取得することができる。具体的には、このモジュールが提供する関数 *get_localzone*, *get_localzone_name* を使用する。

例. 日本国内の計算機環境でローカルタイムゾーンを調べる

```
>>> import tzlocal Enter    ←モジュールの読み込み
>>> tzlocal.get_localzone() Enter    ←ローカルタイムゾーンの取得 (1)
zoneinfo.ZoneInfo(key='Asia/Tokyo')    ←得られたタイムゾーン (1)
>>> tzlocal.get_localzone_name() Enter    ←ローカルタイムゾーンの取得 (2)
'Asia/Tokyo'    ←得られたタイムゾーン (2)
```

この例からわかるように、*get_localzone* 関数は前述の *ZoneInfo* クラスのオブジェクトとして、*get_localzone_name*

¹⁸³Python の標準ライブラリではないので、OS のコマンド「`pip install tzlocal`」を実行するなどして、当該計算機環境にインストールする必要がある。公式インターネットサイト：<https://pypi.org/project/tzlocal/>

関数は文字列としてローカルタイムゾーンを返す。

4.1.2 time モジュール

先に説明した `datetime` モジュールと異なり、**UNIX 時間**に基づいて日付や時刻を取り扱うものとして `time` モジュールがある。UNIX 時間とは、1970 年 1 月 1 日 0 時 0 分 0 秒 (UTC)¹⁸⁴ からの秒数であり**エポック秒** (seconds since the epoch) と呼ぶこともある。

`time` モジュールが提供する関数は、UNIX 環境の C 言語の API に由来するものが多い。

このモジュールは使用に先立って、次のようにしてシステムに読み込んでおく必要がある。

```
import time
```

4.1.2.1 基本的な機能

■ 現在のエポック秒の取得

`time` 関数で現在のエポック秒を取得することができる。

例. 現在のエポック秒の取得

```
>>> import time  [Enter]    ←モジュールの読み込み
>>> t = time.time()  [Enter]    ←現在のエポック秒の取得
>>> print(t,'(sec)', type:',type(t))  [Enter]    ←値とその型を調べる
1690616550.793217 (sec), type:  <class 'float'>    ←浮動小数点数である
```

■ 日付、時刻の情報の取得

エポック秒を年、月、日、時、分、秒などの情報から成る `struct_time` オブジェクトに変換するには `gmtime`, `localtime` 関数を使用する。(前者は UTC、後者はローカルタイムである) これら関数を引数なしで実行すると、その時点のエポック秒に対する `struct_time` オブジェクトを返す。

先の例で得られた変数 `t` の値を `struct_time` オブジェクトに変換する例を次に示す。

例. エポック秒を `struct_time` に変換する (先の例の続き)

```
>>> time.gmtime( t )  [Enter]    ← UTC の解釈で変換
time.struct_time(tm_year=2023, tm_mon=7, tm_mday=29, tm_hour=7,    ← struct_time オブジェクト
tm_min=42, tm_sec=30, tm_wday=5, tm_yday=210, tm_isdst=0)
>>> time.localtime( t )  [Enter]    ←ローカルタイムの解釈で変換
time.struct_time(tm_year=2023, tm_mon=7, tm_mday=29, tm_hour=16,    ← struct_time オブジェクト
tm_min=42, tm_sec=30, tm_wday=5, tm_yday=210, tm_isdst=0)
```

`struct_time` オブジェクトの内部の要素はインデックス (スライス) を与えて参照することができる。次の例は、`struct_time` オブジェクトをイテラブルとして `for` 文で扱うものである。

例. インデックスを指定して `struct_time` オブジェクトの内部の要素を参照する (先の例の続き)

```
>>> for i,e in enumerate(time.localtime( t )):  [Enter]    ←反復の記述 (ここから)
...     print('st['+str(i)+']='+str(e)+' ',sep='',end='')
... else: print()
...  [Enter]    ←反復の記述 (ここまで)
st[0]=2023 st[1]=7 st[2]=29 st[3]=16 st[4]=42 st[5]=30 st[6]=5 st[7]=210 st[8]=0    ←出力
```

`struct_time` オブジェクトの内部の要素は属性を指定して参照することもできる。

例. `struct_time` オブジェクトの「年」の属性 `tm_year` の参照 (先の例の続き)

```
>>> st = time.localtime( t )  [Enter]    ← struct_time オブジェクトを取得し
>>> st.tm_year  [Enter]    ←「年」の属性 tm_year を参照する
2023    ←「年」の値が得られている
```

`struct_time` オブジェクトの各属性を表 32 に示す。

¹⁸⁴この時刻を **UNIX エポック** (the epoch) と呼ぶ。

表 32: struct_time オブジェクトの属性

インデックス	属性	解説
0	tm_year	年
1	tm_mon	月 ([1,12] の間の数)
2	tm_mday	日 ([1,31] の間の数)
3	tm_hour	時 ([0,23] の間の数)
4	tm_min	分 ([0,59] の間の数)
5	tm_sec	秒 ([0,61] の間の数)
6	tm_wday	曜日 ([0,6] の間の数, 月曜が 0)
7	tm_yday	年内の日 ([1,366] の間の数)
8	tm_isdst	夏時間 (無効:0, 有効:1, 不明:-1)
N/A	tm_zone	タイムゾーンの短縮名 (タイムゾーン ID ではない)
N/A	tm_gmtoff	UTC から東方向へのオフセット (秒)

注) tm_sec は整数値であり、小数点以下は無い

属性 tm_zone の値はタイムゾーン ID ではない。(次の例参照)

例. tm_zone 属性の参照 (先の例の続き)

```
>>> st.tm_zone
'東京 (標準時)'
```

←このような表現となっている

■ 日付, 時刻の値からエポック秒を得る方法

ローカルタイムを表す struct_time オブジェクトをエポック秒に変換するには mktime 関数を用いる。

例. struct_time オブジェクトをエポック秒に変換する (先の例の続き)

```
>>> time.mktime( st )
1690616550.0
```

←先の st をエポック秒に変換する
←エポック秒 (小数点以下が切り捨てられていることに注意)

mktime 関数の引数には 9 個の要素 (struct_time の各要素に対応する) を持つタプルを与えることもできる。

例. タプルをエポック秒に変換する (先の例の続き)

```
>>> time.mktime( (2023,7,29,16,42,30,5,210,0) )
1690616550.0
```

←引数にタプルを与える
←エポック秒 (小数点以下が切り捨てられていることに注意)

参考) UTC を表す struct_time オブジェクトをエポック秒に変換するには calendar モジュールの timegm 関数を用いる。

例. UTC の struct_time オブジェクトをエポック秒に変換する

```
>>> import time
>>> t = time.time()
>>> t
1690702599.2582412
>>> g = time.gmtime( t )
>>> import calendar
>>> calendar.timegm( g )
1690702599
```

← time モジュールの読み込み
←現在のエポック秒を取得
←確認
←浮動小数点数
←それを UTC の struct_time に変換
← calendar モジュールの読み込み
← UTC の struct_time をエポック秒に変換
←整数値

timegm 関数の戻り値が整数値であることに注意。

4.1.2.2 日付, 時刻を表す文字列表現

■ エポック秒を可読な文字列に変換する関数 ctime

書き方: ctime(t)

エポック秒 t をローカルタイム解釈で '曜日 月 日 時:分:秒 年' の文字列に変換したものを返す。

例. ctime による可読な文字列への変換

```
>>> import time      [Enter]      ←モジュールの読み込み
>>> t = time.time()   [Enter]      ←現在のエポック秒を取得
>>> time.ctime( t )   [Enter]      ←それを文字列に変換
'Sun Jul 30 12:30:19 2023'        ←変換結果
```

■ struct_time の内容を可読な文字列に変換する関数 asctime

書き方: asctime(st)

struct_time オブジェクト st の内容を '曜日 月 日 時:分:秒 年' の文字列に変換したものを返す。

例. asctime による可読な文字列への変換 (先の例の続き)

```
>>> st = time.localtime( t ) [Enter] ←エポック秒をローカルタイムの struct_time オブジェクトに変換
>>> time.asctime( st ) [Enter] ←上で得た struct_time オブジェクトを文字列にに変換
'Sun Jul 30 12:30:19 2023'        ←変換結果
```

参考) ctime, asctime 関数が返す文字列の書式は, C 言語における同名の関数に由来する。

■ struct_time の内容を書式に沿った文字列に変換する関数 strftime

書き方: strftime(fmt, st)

書式文字列 fmt に沿った形で struct_time オブジェクト の内容を文字列に変換して返す。書式文字列は datetime モジュールの同名のメソッドに与えるもの (p.233 の表 31) と概ね同じである。

例. strftime による書式整形 (先の例の続き)

```
>>> s = time.strftime( '%Y/%m/%d(%a) %H:%M:%S', st ) [Enter] ←先に得た st を書式整形
>>> s [Enter]      ←確認
'2023/07/30(Sun) 12:30:19'        ←変換結果
```

この関数は第 2 引数を省略すると, ローカルタイムの現在時刻を書式整形する。

■ 文字列表現の日付, 時刻を struct_time オブジェクトに変換する関数 strptime

書き方: strptime(文字列, fmt)

日付, 時刻を表現した「文字列」を書式文字列 fmt (先の strftime と同様) に沿った形で解釈し, それを struct_time オブジェクトに変換して返す。

例. strptime による文字列の読み取り (先の例の続き)

```
>>> time.strptime( s, '%Y/%m/%d(%a) %H:%M:%S' ) [Enter] ←上の例で得た文字列を読み取る
time.struct_time(tm_year=2023, tm_mon=7, tm_mday=30, tm_hour=12, ←得られた
tm_min=30, tm_sec=19, tm_wday=6, tm_yday=211, tm_isdst=-1)      struct_time オブジェクト
```

この関数は第 2 引数 (書式文字列) を省略すると, ctime, asctime 関数が返す文字列の書式が適用される。

例. 書式文字列を省略した場合の strptime 関数の動作 (先の例の続き)

```
>>> s = time.ctime( t ) [Enter]      ← ctime 形式の文字列の作成
>>> s [Enter]      ←内容確認
'Sun Jul 30 12:30:19 2023'        ←これをそのまま
>>> time.strptime( s ) [Enter]      ← struct_time オブジェクトに変換する
time.struct_time(tm_year=2023, tm_mon=7, tm_mday=30, tm_hour=12, ←得られた
tm_min=30, tm_sec=19, tm_wday=6, tm_yday=211, tm_isdst=-1)      struct_time オブジェクト
```

4.1.2.3 応用例: datetime オブジェクトとの間の変換

time モジュールの strftime 関数と strptime 関数, datetime モジュールの strftime メソッドと strptime メソッドを応用すると, struct_time オブジェクトと datetime オブジェクトの間の相互の変換が実現できる。

■ datetime オブジェクト → struct_time オブジェクトの変換の例

例. datetime オブジェクトの作成

```
>>> from datetime import datetime [Enter] ← datetime クラスの読み込み
>>> d = datetime.now() [Enter] ← 現在時刻の datetime オブジェクトを作成
>>> d [Enter] ← 確認
datetime.datetime(2023, 7, 30, 17, 8, 19, 372091)
```

変数 d に現在時刻の datetime オブジェクトが得られている。次にこれを文字列に変換し、更にそれを struct_time オブジェクトに変換する。

例. datetime オブジェクト → 文字列の変換（先の例の続き）

```
>>> s = d.strftime('%Y/%m/%d %H:%M:%S') [Enter] ← datetime を文字列に変換
>>> s [Enter] ← 確認
'2023/07/30 17:08:19' ← 文字列になっている
```

例. 文字列 → struct_time オブジェクトの変換（先の例の続き）

```
>>> import time [Enter] ← time モジュールの読み込み
>>> st = time.strptime(s, '%Y/%m/%d %H:%M:%S') [Enter] ← 上で作成した文字列を struct_time に変換
>>> st [Enter] ← 確認
time.struct_time(tm_year=2023, tm_mon=7, tm_mday=30, tm_hour=17, ← struct_time オブジェクト
tm_min=8, tm_sec=19, tm_wday=6, tm_yday=211, tm_isdst=-1) になっている
```

■ struct_time オブジェクト → datetime オブジェクトの変換の例

例. struct_time → 文字列 → datetime オブジェクト（先の例の続き）

```
>>> s = time.strftime('%Y/%m/%d %H:%M:%S', st) [Enter] ← struct_time を文字列に変換
>>> s [Enter] ← 確認
'2023/07/30 17:08:19'
>>> datetime.strptime(s, '%Y/%m/%d %H:%M:%S') [Enter] ← 上で作成した文字列を datetime に変換
datetime.datetime(2023, 7, 30, 17, 8, 19) ← 得られた datetime オブジェクト
```

注) 実際の処理ではタイムゾーンに配慮すること。

4.1.2.4 時間の計測

プログラムの実行にかかった時間を計測するには、開始時点での時刻と終了時点での時刻の値の差を取るという方法がある。2 を 500,000 回掛け算することで $2^{500,000}$ を求めるプログラム test10-1.py を示す。このプログラムは計算にかかった時間を出力する。

プログラム：test10-1.py

```
1 # coding: utf-8
2 import time      # 必要なモジュールの読み込み
3
4 t1 = time.time()
5 n = 1
6 for i in range(500000):
7     n *= 2
8 t2 = time.time()
9
10 print(t2-t1, '(sec)')
11 #import sys; sys.set_int_max_str_digits(40000); print(n)
```

このプログラムを実行すると、例えば

3.1404953002929688 (sec)

などと、実行に要した時間¹⁸⁵が表示される。また、最終行のコメント '#' を外すと $2^{500,000}$ の計算結果が表示¹⁸⁶される。

¹⁸⁵当然であるが、使用する計算機環境によって実行時間は異なる。

¹⁸⁶この部分の処理に関しては「2.4.6.2 巨大な整数値を扱う際の注意」(p.13)を参照のこと。

プログラムの実行時間の計測には、後に説明する `timeit` モジュールを使用する方法もある。

■ 更に正確な時間計測

`time.time()` の差分計算による経過時間の計測結果は**実時間**である。実際の計算機システムでは、複数のスレッドやプロセスが同時に実行されるので、個々のスレッドやプロセスが CPU を実際に使用した時間（**CPU 時間**）と実時間は異なる。当該スレッドが起動した時点をも 0 として、当該スレッドが使用した CPU 時間を得るには `thread.time` 関数を使用する。また、当該プロセスが起動した時点をも 0 として、当該プロセスが使用した CPU 時間を得るには `process.time` 関数を用いる。これらの関数を用いることで、スレッドやプロセスが実際に実行に要した時間を算出することができる。

※ スレッドとプロセスに関しては、後の「4.4 マルチスレッドとマルチプロセス」(p.255) で詳しく解説する。

先のサンプルプログラム `test10-1.py` と同様の処理を行い、スレッドとしての CPU 時間を計測する例を `test10-1.thread.py` に、プロセスとしての CPU 時間を計測する例を `test10-1.process.py` に示す。

プログラム：test10-1.thread.py

```
1 # coding: utf-8
2 import time
3
4 t1 = time.thread_time()
5 n = 1
6 for i in range(500000):
7     n *= 2
8 t2 = time.thread_time()
9
10 print(t2-t1, '(sec)')
```

このプログラムを実行すると、例えば

2.984375 (sec)

などと、スレッドとしての実行に要した CPU 時間が表示される。

プログラム：test10-1.process.py

```
1 # coding: utf-8
2 import time
3
4 t1 = time.process_time()
5 n = 1
6 for i in range(500000):
7     n *= 2
8 t2 = time.process_time()
9
10 print(t2-t1, '(sec)')
```

このプログラムを実行すると、例えば

3.09375 (sec)

などと、プロセスとしての実行に要した CPU 時間が表示される。

▲注意▲

マルチスレッドプログラミングにおいては、個々のサブスレッド内で得られる `thread.time` 関数の値が、当該サブスレッドが実際に実行した処理の CPU 時間を反映しないことがあるので注意すること。それについて次のサンプル `test10-2.thread.py` で確認する。

プログラム：test10-2.thread.py

```
1 # coding: utf-8
2 import time
3 from threading import Thread
4
5 # 実行時間がかかる処理
6 def longTask(res, i):
7     ts = time.thread_time() # スレッド開始時間
8     n = 1
9     for _ in range(500000):
10         n *= 2
11     te = time.thread_time() # スレッド終了時間
12     res[i] = te - ts # 実行時間の格納
13
14 R = [0, 0] # 処理結果受領用リスト
15
16 # スレッド作成
17 th1 = Thread(target=longTask, args=(R, 0))
18 th2 = Thread(target=longTask, args=(R, 1))
19
20 # スレッド実行
21 t1 = time.time()
22 th1.start(); th2.start() # スレッドの開始
23 th1.join(); th2.join() # スレッド終了の待機
24 t2 = time.time()
25
```

```

26 # 測定結果の報告
27 print('メインスレッドの経過時間 (秒):', t2-t1)
28 print('スレッド th1 の CPU 時間 (秒):', R[0])
29 print('スレッド th2 の CPU 時間 (秒):', R[1])

```

これは先の test10-1.thread.py と同様の処理を 2 つのサブスレッドで同時に実行するものである。これを実行すると次のような結果が表示される。

例. test10-2.thread.py 実行後のコンソール出力

```

メインスレッドの経過時間 (秒):  6.486693382263184
スレッド th1 の CPU 時間 (秒):  0.78125
スレッド th2 の CPU 時間 (秒):  0.984375

```

メインスレッドの経過時間は先の test10-1.thread.py の 2 倍程度 (同じ処理を 2 つ同時に実行しているため) となっているが、個々のサブスレッド th1, th2 の CPU 時間が非常に小さな値となっている。この値は、個々のサブスレッドが実際に行っている処理の CPU 時間を反映していないことに注意すること。

4.1.2.5 プログラムの実行待ち

time モジュールの sleep 関数を用いると、指定した時間プログラムを待機させる¹⁸⁷ ことができる。

書き方: sleep(待ち時間)

CPU 時間を消費せずにプログラムの実行を「待ち時間」(単位: 秒) だけ待機する。

プログラム: timeSleep01.py

```

1 # coding: utf-8
2 import time
3
4 print('プログラムの開始')
5 time.sleep(3.0)
6 print('プログラムの終了')

```

これを実行すると次のような出力が得られる。

例. timeSleep01.py を実行した際のコンソール出力

```

プログラムの開始          ←この出力の 3 秒後に
プログラムの終了          ←これが出力される

```

4.1.3 timeit モジュール

プログラムの実行時間の計測には timeit モジュールを使用する方法¹⁸⁸ がある。このモジュールを使用するには import timeit などとして Python 処理系に読み込む必要がある。

このモジュールには同名の関数 timeit が提供されており、これを用いて文字列として与えたプログラムの実行時間を計測することができる。

書き方: timeit(プログラム, number=繰り返し回数, globals=グローバルオブジェクトの辞書)

「プログラム」に文字列で記述した Python のプログラムを与える。「繰り返し回数」には与えたプログラムを実行する回数を指定する。(暗黙で 1,000,000 が設定されているので注意すること)「グローバルオブジェクトの辞書」には、実行するプログラムに必要なグローバルオブジェクトの辞書(変数名に対する値の辞書)を与える。この関数は実行に要した時間(秒単位の浮動小数点数)¹⁸⁹ を返す。

注意) 引数に globals=globals() を指定した場合も、timeit 関数で実行されたプログラムの影響は、実際のグローバル変数には反映されない。これは exec, eval 関数の場合と異なるので注意すること。

以下にサンプルを示して timeit 関数の使用方法について説明する。

¹⁸⁷厳密には当該スレッドを待機させる。スレッドに関しては後の「4.4 マルチスレッドとマルチプロセス」(p.255)を参照のこと。

¹⁸⁸time モジュールを使用する方法よりも正確な実行時間が得られる。

¹⁸⁹経過時間 (elapsed time, wall clock time)

例. サンプル用の関数定義

```
>>> def p2n(n): Enter    ← 2n を計算する関数
...     r = 1 Enter
...     for i in range(n): r *= 2 Enter
...     return r Enter
... Enter    ←関数定義の記述の終了
>>>    ← Python のプロンプトに戻った
```

この関数 p2n の実行時間を計測する例を次に示す.

例. プログラムの実行時間の計測 (先の例の続き)

```
>>> import timeit Enter    ←モジュールの読み込み
>>> prg = 'p2n(10000)' Enter    ←実行するプログラムを文字列として用意
>>> timeit.timeit(prg,number=100,globals=globals()) Enter ← 100 回実行してその時間を計測
0.19903010001871735    ←実行にかかった時間 (秒)
>>> timeit.timeit(prg,number=1000,globals=globals()) Enter ← 1000 回実行してその時間を計測
1.8931540999910794    ←実行にかかった時間 (秒)
```

timeit 関数の引数 'globals=' に与える globals() はグローバルオブジェクトの辞書を取得する関数で、これに関しては後の「4.20 使用されているシンボルの調査」(p.339) で解説する. 引数 'number=' には反復実行の回数を与える.

4.1.3.1 計測方法の指定

Timer オブジェクトを用いると、プログラムの実行時間の計測をスレッドの時間、あるいはプロセスの時間として計測することができる.

書き方: `timeit.Timer(計測対象プログラム, globals=グローバルオブジェクト辞書, timer=計測用関数)`

「計測対象プログラム」(文字列) の実行時間を「計測用関数」(引数なし) を用いて計測するための Timer オブジェクトを返す. 「計測用関数」には、先の「更に正確な時間計測」(p.241) で解説した `time.thread_time`, `time.process_time` が使える. (デフォルトは `time.perf_counter`)

Timer オブジェクトに timeit メソッドを実行することで実行時間が得られる.

書き方: `Timer オブジェクト.timeit(number=反復実行の回数)`

このメソッドは計測時間 (秒) を返す.

例. スレッドの実行時間、プロセスの実行時間の計測 (先の例の続き)

```
>>> import time Enter    ←モジュールの読み込み
>>> tmr1 = timeit.Timer(prg,globals=globals(),timer=time.thread_time) Enter ←タイマー生成
>>> tmr1.timeit(number=1000) Enter    ←時間 (スレッド) 計測実行 (1000 回反復)
1.9375    ←計測結果 (秒)
>>> tmr2 = timeit.Timer(prg,globals=globals(),timer=time.process_time) Enter ←タイマー生成
>>> tmr2.timeit(number=1000) Enter    ←時間 (プロセス) 計測実行 (1000 回反復)
1.765625    ←計測結果 (秒)
```


4.2 ロケール (locale)

日付や時刻の書式、通貨の書式、数値の書式などは国や言語圏毎に異なることがあり、アプリケーションプログラムを作成する際は、それらをロケールとして識別することがある。ロケールは言語コード (ISO 639-1) と国コード (ISO 3166-1) から成るものであり、例えば日本のロケールは、日本語を意味する言語コード「ja」と日本国を意味する国コード「JP」を用いて「ja_JP」などと表記される。更にロケールにはエンコーディングの情報を持たせることも可能で、先の例に「utf-8」の情報を加えるとロケールは「ja_JP.UTF-8」となる。また、ロケールの記述に用いる言語コード、国コード、エンコーディングの表記は大文字／小文字の違いなど様々なバリエーションがある。

Python 言語処理系においては、ロケールの設定などの機能を locale モジュールが提供している。実際に、datetime モジュールや time モジュールなどのいくつかの API がロケールの設定状態の影響を受ける。例えば次のような datetime オブジェクトの書式整形について考える。

例. datetime オブジェクトの書式整形

```
>>> from datetime import datetime, timedelta, timezone  Enter    ←モジュールの読み込み
>>> z = timezone(timedelta(hours=9))  Enter    ←日本のタイムゾーン
>>> d = datetime(2023,7,31,13,15,23,tzinfo=z)  Enter    ←datetime オブジェクトの作成
>>> d.strftime('%Y/%m/%d (%a:%A) (%b:%B) %H:%M:%S')  Enter    ←書式整形
'2023/07/31 (Mon:Monday) (Jul:July) 13:15:23'  ←結果
```

この例では %a, %A によって曜日、%b, %B によって月名を得ているが英語圏の書式となっていることがわかる。(実際に多くの計算機環境において、この状態がデフォルトとなっている)

次に、ロケールを他の言語圏に変更して同様の処理を行う例を示す。

例. ロケールの設定変更 (先の例の続き)

```
>>> import locale  Enter    ←モジュールの読み込み
>>> locale.setlocale( locale.LC_TIME, 'ja_JP' )  Enter    ←時間のロケールを日本に設定
'ja_JP'  ←変更された
>>> d.strftime('%Y/%m/%d (%a:%A) (%b:%B) %H:%M:%S')  Enter    ←書式整形
'2023/07/31 (月:月曜日) (7:7 月) 13:15:23'  ←日本語になっている
>>> locale.setlocale( locale.LC_TIME, 'fr_FR' )  Enter    ←時間のロケールをフランスに設定
'fr_FR'  ←変更された
>>> d.strftime('%Y/%m/%d (%a:%A) (%b:%B) %H:%M:%S')  Enter    ←書式整形
'2023/07/31 (lun.:lundi) (juil.:juillet) 13:15:23'  ←フランス語になっている
```

この例で示した setlocale 関数がロケールを設定するものである。

書き方: setlocale(カテゴリ, ロケール)

ロケールを設定する際にはカテゴリ (表 33) を指定する

表 33: ロケールのカテゴリ

カテゴリ	解説	影響範囲 (備考)
LC.ALL	全てのロケール設定を総合したもの	
LC.COLLATE	文字列の並べ替えに関する設定	locale モジュールの strcoll, strxfrm 関数
LC.CTYPE	文字タイプ関連の設定	string モジュール
LC.MONETARY	金額などの値の書式化に関する設定	
LC.NUMERIC	数値の文字列表現に関する設定	locale モジュールの format, atoi, atof, str 関数
LC.TIME	時刻の書式化に関する設定	datetime, time モジュール
LC.MESSAGES	メッセージ表示に関する設定	サポートしていない処理系もある。

実際にはこの表のカテゴリを「locale. カテゴリ」のように記述する。

4.2.1 ロケール設定の確認

getlocale 関数によってロケールの設定を確認することができる。

書き方: getlocale(カテゴリ)

指定した「カテゴリ」の設定状態を ('言語コード 国コード', エンコーディング) のタプルとして返す。

Python 処理系を起動した直後のロケール設定を確認するプログラム locale01.py を示す。

プログラム：locale01.py

```
1 # coding: utf-8
2 import locale
3
4 # locale.setlocale( locale.LC_ALL, '' )      # OSのロケール設定に準拠する
5 # locale.setlocale( locale.LC_ALL, 'C' )     # ANSI C(POSIX)のロケール設定に準拠する
6
7 cat = locale.LC_COLLATE
8 print( 'LC_COLLATE (', cat, '): ', locale.getlocale(cat), sep='' )
9 cat = locale.LC_CTYPE
10 print( 'LC_CTYPE (', cat, '): ', locale.getlocale(cat), sep='' )
11 cat = locale.LC_MONETARY
12 print( 'LC_MONETARY (', cat, '): ', locale.getlocale(cat), sep='' )
13 cat = locale.LC_NUMERIC
14 print( 'LC_NUMERIC (', cat, '): ', locale.getlocale(cat), sep='' )
15 cat = locale.LC_TIME
16 print( 'LC_TIME (', cat, '): ', locale.getlocale(cat), sep='' )
```

このプログラムの実行による出力の例を次に示す。

```
LC_COLLATE (1): (None, None)
LC_CTYPE (2): ('Japanese_Japan', '932')
LC_MONETARY (3): (None, None)
LC_NUMERIC (4): (None, None)
LC_TIME (5): (None, None)
```

このプログラムの4行目の先頭にあるコメント記号「#」を外すと、Python 処理系のロケールがOSのロケール設定に準拠したものとなる。また5行目の先頭にあるコメント記号「#」を外すとANSI C(POSIX)に準拠した設定となるので試みられたい。

参考) ロケールを使用するアプリケーションにおいては `locale.setlocale(locale.LC_ALL, '')` を実行しておくことが望ましい。

4.2.2 使用できるロケールの調査

ロケールの表記を始めとする各種の情報は辞書の形で `locale.locale_alias` に保持されている。この中から **言語コード** **国コード** の形式になっているものを取り出すことで、Python 言語処理系で使われるロケールを調べることができる。(下記プログラム例：locale02.py)

プログラム：locale02.py

```
1 # coding: utf-8
2 import locale
3
4 b = set()
5 for e in locale.locale_alias.keys():      # ロケール収集
6     if '_' in e and '.' not in e and len(e)<6:
7         b.add(e)
8
9 for i,L in enumerate(sorted(list(b))):    # 整列して出力
10     if (i+1)%18!=0:
11         print(f'{L:6s}',end='')
12     else:
13         print(L)
14 else: print()
```

このプログラムの実行による出力の例を次に示す。

```
a3_az aa_dj aa_er aa_et af_za ak_gh am_et an_es ar_aa ar_ae ar_bh ar_dz ar_eg ar_in ar_iq ar_jo ar_kw ar_lb
ar_ly ar_ma ar_om ar_qa ar_sa ar_sd ar_ss ar_sy ar_tn ar_ye as_in az_az az_ir be_by bg_bg bi_vu bn_bd bn_in
bo_cn bo_in br_fr bs_ba c_c ca_ad ca_es ca_fr ca_it ce_ru cs_cs cs_cz cv_ru cy_gb cz_cz da_dk de_at de_be
de_ch de_de de_it de_lu dv_mv dz_bt ee_ee el_cy el_gr en_ag en_au en_be en_bw en_ca en_dk en_gb en_hk en_ie
en_il en_in en_ng en_nz en_ph en_sg en_uk en_us en_za en_zm en_zw eo_eo eo_xx es_ar es_bo es_cl es_co es_cr
:
(以下省略)
:
```

4.2.3 その他

4.2.3.1 現在のエンコーディングの調査

getencoding 関数を使用すると現在のエンコーディングを調べることができる。

例. エンコーディングを調べる

```
>>> import locale Enter      ←モジュールの読み込み
>>> locale.setlocale( locale.LC_ALL, '' ) Enter      ←ロケールの初期化
'Japanese_Japan.932'                      ←このようなロケールに設定された
>>> locale.getencoding() Enter      ←エンコーディングを調べる
'cp932'                                    ←現在のエンコーディング
```

4.2.3.2 通貨の書式整形

currency 関数を使用すると、数値を通貨の値と見做して書式整形ができる。

書き方： `currency(数値, grouping=桁区切りの有無, international=国際表現にするか否か)`

「数値」を金額表現の文字列に変換して返す。「grouping=」、「international=」には真理値を与える。

例. 日本円による書式整形

```
>>> import locale Enter      ←モジュールの読み込み
>>> locale.setlocale( locale.LC_MONETARY, 'ja_JP' ) Enter      ←通貨を日本円に設定
'ja_JP'                                    ←設定結果
>>> locale.currency( 100000 ) Enter      ←数値を通貨表現の文字列に変換
'¥100000'                                  ←変換結果
>>> locale.currency( 100000, grouping=True ) Enter      ←コンマ区切りありで変換
'¥100,000'                                ←変換結果
>>> locale.currency( 100000, grouping=True, international=True ) Enter
'JPY100,000'                              ←変換結果          ↑コンマ区切りありの国際表現で変換
```

例. 米ドルによる書式整形（先の例の続き）

```
>>> locale.setlocale( locale.LC_MONETARY, 'en_US' ) Enter      ←通貨を米ドルに設定
'en_US'                                    ←設定結果
>>> locale.currency( 100000 ) Enter
'$100000.00'                              ←変換結果
>>> locale.currency( 100000, grouping=True ) Enter
'$100,000.00'                              ←変換結果
>>> locale.currency( 100000, grouping=True, international=True ) Enter
'USD100,000.00'                          ←変換結果
```

本書で取り上げなかったロケール関連の機能については、Python の公式インターネットサイトなどを参照のこと。

4.3 文字列検索と正規表現

re モジュールを用いることで、高度なパターン検索が実現できる。このモジュールは次のようにして読み込む。

例. モジュールの読み込み

```
>>> import re
```

以後、re モジュールが読み込まれている前提で解説する。

4.3.1 パターンの検索

re クラスの search 関数を使用すると、文字列に含まれるパターンを検索することができる。具体的には、長いテキスト（文字列）の中に、探したい検索キー（文字列）がどの位置にあるかを調べる。

《 文字列の検索 (1) 》

書き方 (1): re.search(検索キー, テキスト)

検索対象の「テキスト」の中から指定した「検索キー」（パターン）が最初に現れる場所を見つけ出す。
「検索キー」は raw 文字列で与える。

書き方 (2): コンパイル済み検索キー.search(テキスト)

「コンパイル済み検索キー」は re クラスの compile 関数で生成された検索キーであり、
これを用いることで検索処理が最適化される。

検索キーのコンパイル: re.compile(検索キー)

この結果、コンパイル済み検索キーが返される。

検索処理が終わると、検索結果を保持する match オブジェクトが返される。テキストの中に検索キーに一致するものが無ければ None が返される。

検索キーには正規表現（後述）を指定することができる。そのため、raw 文字列で与える必要がある。

テキスト 'My name is Taro. I am 19 years old.' の中から 'Taro' の位置を探す例（書き方 (1) の例）を示す。

例. 検索処理

```
>>> import re      Enter      ←モジュールの読み込み
>>> txt = 'My name is Taro. I am 19 years old.' Enter      ←テキストの生成
>>> ptn = r'Taro'   Enter      ←検索キーの生成
>>> res = re.search(ptn,txt) Enter      ←検索の実行
>>> res.span()      Enter      ←検出位置を調べる
(11, 15)           ← 11~14 番目に'Taro'が存在することがわかる
```

この例では、検索結果が変数 res に match オブジェクトとして得られている。それに対して span メソッドを実行することで、検出位置のタプル (n_1, n_2) が得られる。これは、対象テキストの $n_1 \sim n_2 - 1$ のインデックスの位置にパターンが見つかったことを意味する。

検索キーをコンパイルして同様の処理を行ったものが次の例（書き方 (2) の例）である。

例. パターンをコンパイルして検索する例

```
>>> p = re.compile(ptn) Enter      ←検索キーのコンパイル
>>> res = p.search(txt) Enter      ←検索の実行
>>> res.span()      Enter      ←検出位置を調べる
(11, 15)           ← 11~14 番目に'Taro'が存在することがわかる
```

search 関数は、テキストの中で最初に検索キーが現れる場所を探す。同じ検索キーがテキストの中に複数含まれる場合は finditer 関数を使用することで全ての検出位置を取得することができる。

《 文字列の検索 (2) 》

書き方 (3): `re.finditer(検索キー, テキスト)`

検索対象のテキストの中から指定した検索キー (パターン) が現れる箇所を全て見つけ出す。
「検索キー」は **raw 文字列** で与える。

書き方 (4): `コンパイル済み検索キー.finditer(テキスト)`

「コンパイル済み検索キー」は `re` クラスの `compile` 関数で生成された検索キーであり、
これを用いることで検索処理が最適化される。

検索処理が終わると、検索結果を保持する **match オブジェクトのイテレータ** が返される。

次に、`finditer` 関数を用いた検索の例を示す。test11.txt のようなテキストからキーワード「Python」を全て見つけ出す処理の例 (書き方 (4) の例) である。

検索対象のテキスト: test11.txt

```
1 (フリー百科事典「ウィキペディア」より)
2
3 コードを単純化して可読性を高め、読みやすく、また書きやすくしてプログラマの
4 作業性とコードの信頼性を高めることを重視してデザインされた、汎用の高水準言語
5 である。反面、実行速度はCなどの低級言語に比べて犠牲にされている。
6
7 核となる文法 (シンタックス) および意味 (セマンティクス) は必要最小限に抑え
8 られている。その反面、豊富で大規模な文書 (document) や、さまざまな領域に
9 対応する大規模な標準ライブラリやサードパーティ製のライブラリが提供されている。
10 またPythonは多くのハードウェアとOS (プラットフォーム) に対応しており、複数の
11 プログラミングパラダイムに対応している。Pythonはオブジェクト指向、命令型、
12 手続き型、関数型などの形式でプログラムを書くことができる。動的型付け言語であり、
13 参照カウントベースの自動メモリ管理 (ガベージコレクタ) を持つ。
14
15 これらの特性により、PythonはWebアプリケーションやデスクトップアプリケーション
16 などの開発はもとより、システム用の記述 (script) や、各種の自動処理、理工学や
17 統計・解析など、幅広い領域における支持を得る、有力なプログラム言語となった。
18 プログラミング作業が容易で能率的であることは、ソフトウェア企業にとっては
19 投入人員の節約、開発時間の短縮、ひいてはコスト削減に有益であることから、
20 産業分野でも広く利用されている。Googleなど主要言語に採用している企業も多い。
```

このテキストの中から検索キー 'Python' を全て探し出すプログラムを test11.py に示す。

プログラム: test11.py

```
1 # coding: utf-8
2 import re      # 必要なモジュールの読み込み
3
4 # ファイルの内容を一度で読み込む
5 f = open('test11.txt', 'r', encoding='utf-8')
6 text = f.read()
7 f.close()
8
9 # 検索処理
10 ptn = r'Python'      # 検索キー
11 p = re.compile(ptn) # 検索キーのコンパイル
12 res = p.finditer(text)
13
14 # 処理結果の報告
15 print('【検索キー"Python"の検索結果】')
16 print('-----')
17 n = 0
18 for i in res:
19     n += 1
20     print(n, '番目の検出位置: ', i.span())
21 print('-----')
22 print(n, '個検出しました。')
```

テキスト中には 'Python' という語は複数あり、得られるイテレータは、検索が該当した回数分の `match` オブジェクトを含んでいる。そこで、18~20行目のように `for` 文で1要素ずつ取り出している。取り出された要素に対して `span`

メソッドを実行することで、テキスト中に見つかった検索キーワードの位置を取得することができる。

このプログラムを実行した結果を次に示す。

【検索キー"Python"の検索結果】

```
-----
1 番目の検出位置: (256, 262)
2 番目の検出位置: (319, 325)
3 番目の検出位置: (424, 430)
-----
```

3 個検出しました。

4.3.1.1 正規表現を用いた検索

検索キーとして、固定された文字列のみを用いるのではなく、指定した条件に一致する部分をテキストの中から見つけ出す場合に**正規表現**が非常に有用である。例えばアルファベットのみから成る文字列を見つけて出す場合について考える。

「アルファベット文字列」を意味するパターンを正規表現で表すと `'[a-zA-Z]+'` となる。(詳しくは後述する) 先のプログラムの 10 行目を

```
ptn = r'[a-zA-Z]+'
```

20 行目を

```
print(n, ' 番目の検出位置: %t', i.span(), '%t', i.group())
```

と書き換えて(タイトル表示部分も変えて)実行すると、次のような結果となる。

【アルファベット文字列の検索結果】

```
-----
1 番目の検出位置: (110, 111)      C
2 番目の検出位置: (193, 201)     document
3 番目の検出位置: (256, 262)     Python
      :
      (途中省略)
      :
8 番目の検出位置: (479, 485)     script
9 番目の検出位置: (631, 637)     Google
-----
```

9 個検出しました。

このように柔軟に検索パターンを構成する場合に正規表現が有用である。検索パターンが具体的にどのような文字列に一致したかを調べるにはメソッド `group` を使用する。

この例では `'[a-zA-Z]+'` で半角アルファベットを表したが、正規表現では `'[~]'` で文字の範囲(集合)を表す。例えば `'[a-z]'` とするとこれは「アルファベット小文字」を意味する。また、`'[A-Z]'` では「アルファベット大文字」を意味する。更に `'+'` は「それらが 1 文字以上続く列」を意味する。従って、

`'[a-zA-Z]+'`

は、

「アルファベット小文字もしくは大文字」かつ「それらが 1 文字以上続く列」

を意味し、「アルファベットの文字列」を意味するパターンとなる。同様に「数字のみの列」を意味する正規表現は `'[0-9]+'` となる。

特定の文字集合、例えば `'c', 'n', 'x'` のどれか」は `'[cnx]'` と記述する。

例. 'a' もしくは 'b' もしくは 'c' に一致する文字の検索

```
>>> print( re.search( r'[abc]', 'abcd' ) )  Enter    ←パターン検索実行
<re.Match object; span=(0, 1), match='a'>    ←先頭の 'a' に一致
```

これはパターン `'[abc]'` を検索するものであるが、括弧 `'[]'` の中に `'^'` を記述すると「それら以外のパターン」を検索する。

例. 「それ以外」のパターン検索

```
>>> print( re.search( r'[^abc]', 'abcd' ) )  ←パターン検索実行
<re.Match object; span=(3, 4), match='d'> ←末尾の 'd' に一致
```

これは「『a' もしくは'b' もしくは'c'』でないもの」の検索であり、「a' でなく、かつ、'b' でなく、かつ、'c' でないもの」と同義¹⁹⁰ である。結果として末尾の 'd' に一致している。

本書では正規表現の全てについての解説はしないが、特に使用頻度が高いと考えられるパターン表現について説明する。

■ 正規表現で構成する文字列のパターン

基本的には「1 文字分のパターン」と「繰り返し」を接続したものである。先に説明した「[~]」が「1 文字分のパターン」、'+」が「繰り返し」を意味する。「1 文字分のパターン」の基本的なものを表 34 に挙げる。

表 34: 正規表現のパターン（一部）

パターン	意味
[c ₁ c ₂ c ₃ ...]	特定の文字の集合 c ₁ c ₂ c ₃ ... （これらの内のどれかに該当）否定の '^' が使える。
[c ₁ -c ₂]	c ₁ ~c ₂ の間に含まれる文字。文字の範囲をハイフン '-' でつなげる。 例: [a-d] → 'a', 'b', 'c', 'd' のどれか。 （否定の '^' が使える）
.	任意の 1 文字
¥d	数字（[0-9] と同じ）
¥D	数字以外
¥s	空白文字（タブ、改行文字なども含む）
¥S	空白文字以外（¥s でないもの）
¥w	アンダースコアを含む半角英数字（[a-zA-Z0-9_] と同じ）
¥W	半角英数字以外（¥w でないもの）

「繰り返し」の基本的なものを表 35 に挙げる。

表 35: 繰り返しの表記（一部）

表記	意味	表記	意味
+	1 回以上	*	0 回以上
?	0 回かもしくは 1 回	{m,n}	m 回以上 n 回以下（回数が多い方を優先）
		{m,n}?	m 回以上 n 回以下（回数が小さい方を優先）
{n}	n 回	{m,}	m 回以上

正規表現のパターンには多バイト文字（全角文字）を使用することもできる。

例. 全角文字列 'かきくけこ' を検索する

```
>>> print( re.search( '[かきくけこ]+', 'あいいうえおかきくけこさしすせそ' ) ) 
<re.Match object; span=(5, 10), match='かきくけこ'> ←検出できている
```

■ 文字コードを検索パターンに使用する方法

「¥x」に続いて 16 進数の ASCII コードを記述したものを当該文字コードが表す文字の検索パターンとして使用できる。例えばアルファベット小文字を意味する正規表現 '[a-z]' は '¥x61-¥x7a]' と記述することができる。

例. アルファベット小文字の検索パターンを文字コードで記述する

```
>>> txt = r'0123abcDEF#&?'  ←テキスト
>>> ptn = r'¥x61-¥x7a)+'  ←パターン
>>> res = re.search(ptn,txt)  ←検索実行
>>> print( res.span(),res.group() )  ←結果の出力
(4, 7) abc ←インデックス範囲 4 以上 7 未満の位置に 'abc' を検出
```

¹⁹⁰ブール代数や集合論における「ド・モルガン則」でも理解できる。

■ Unicode 文字の検索

Unicode 文字を検索する場合「¥u」に続いて 16 進数のコードポイント値を記述して検索パターンとすることができる。

例. Unicode の平仮名を検索する

```
>>> txt = '0123 あいうえお abc'  Enter    ←テキスト
>>> ptn = r'[\u3041-\u3096]+'      Enter    ←パターン
>>> res = re.search(ptn,txt)        Enter    ←検索実行
>>> print( res.span(),res.group() ) Enter    ←結果の出力
(4, 9) あいうえお                  ←インデックス範囲 4 以上 9 未満の位置に 'あいうえお' を検出
```

Unicode における平仮名のコード範囲は ¥u3041~¥u3096 であり、この例はそれを応用したものである。

4.3.1.2 検索パターンの和結合

複数の検索パターンを「|」で和結合することができる。これを応用すると高度なパターン検索が可能になる。これに関して以下に例を挙げて説明する。

■ サンプルケース：HTML タグのパターン検索

次のような 3 つのパターンのタグ全てに一致する検索処理について考える。

1. tx1 = '<h1>見出し 1</h1>': 単純なタグ
2. tx2 = '<h2 id="hd2">見出し 2</h2>': タグ内に属性などの記述があるもの
3. tx3 = '<h3 >見出し 3</h3>': タグ内に余計な空白があるもの

このような文字列が変数 tx1, tx2, tx3 に設定されているとする。

例. 上記 1 のパターンでそれぞれ検索

```
>>> ptn = r'<h¥d>'  Enter    ←上記 1 のパターン
>>> print( re.search(ptn,tx1) ) Enter    ←上記 1 に対して検索実行
<re.Match object; span=(0, 4), match='<h1>'>    ←先頭のタグに一致
>>> print( re.search(ptn,tx2) ) Enter    ←上記 2 に対して検索実行
None                                              ←一致せず
>>> print( re.search(ptn,tx3) ) Enter    ←上記 3 に対して検索実行
None                                              ←一致せず
```

この例からわかるように「ptn = r'<h¥d>」の検索パターンでは 2 と 3 のパターンに一致させることができない。従って 1, 2, 3 全てのパターンに一致させるには「<h¥d¥s+[^>]*>」というパターンも必要になる。そこで、このパターンと上の例のパターンの和結合による検索を試みる例を次に示す。

例. パターンの和結合

```
>>> ptn = r'<h¥d>|<h¥d¥s+[^>]*>'  Enter    ←パターン和結合
>>> print( re.search(ptn,tx1) ) Enter    ←上記 1 に対して検索実行
<re.Match object; span=(0, 4), match='<h1>'>    ←先頭のタグに一致
>>> print( re.search(ptn,tx2) ) Enter    ←上記 2 に対して検索実行
<re.Match object; span=(0, 13), match='<h2 id="hd2">'>    ←先頭のタグに一致
>>> print( re.search(ptn,tx3) ) Enter    ←上記 3 に対して検索実行
<re.Match object; span=(0, 6), match='<h3 >'>    ←先頭のタグに一致
```

この例のように、検索パターンの結合を応用すると高度な検索が可能となる。

考察) 今回の例の tx1, tx2, tx3 に一致する検索パターンはもっと簡単に記述することができるので考察されたい。

4.3.1.3 正規表現を用いたパターンマッチ

パターンマッチによって、テキストデータから必要な部分を抽出することができる。パターンマッチには match 関数を用いる。

例. テキスト 'uyhtgfdres98234yhnbgtrf' の中から数字の部分のみ取り出す

```
>>> txt = 'uyhtgfdres98234yhnbgtrf'  Enter    ←テキストの生成
>>> ptn = r'[a-zA-Z]+([0-9]+)[a-zA-Z]*'  Enter    ←パターンの生成
>>> res = re.match(ptn,txt)  Enter    ←パターンマッチの実行
>>> res.group(0)  Enter    ←マッチしたか確認
'uyhtgfdres98234yhnbgtrf'    ←マッチしている (テキスト全体)
>>> res.group(1)  Enter    ←パターン ([0-9]+) が一致した部分の取り出し
'98234'    ←数字のみが得られている
```

この例では、検索パターンが '[a-zA-Z]+([0-9]+)[a-zA-Z]*' として与えられている。パターンの中に括弧 '(~)' の部分 (グループ) があるが、抽出したい部分はどのように括弧で括る。

match 関数は、テキストとパターンを一致させるように動作する。すなわちテキストの先頭からパターンを一致させるように試みるものであり、search や finditer による検索とは動作が異なる。

パターンの中には抽出したい括弧付きの部分 (グループ) を複数記述することができ、match が成功すると、得られた match オブジェクトから group メソッドに順番の引数を与えて取り出すことができる。ここに示した例では括弧付きの部分は 1 つであるので、

```
res.group(1)
```

で抽出した部分を得ることができる。

パターンにおけるグループの考え方は検索処理の場合にも当てはまり、search の戻り値の match オブジェクトや finditer の戻り値の各要素の match オブジェクトに対しても 'group(n)' のメソッドを適用することができる。

4.3.1.4 行頭や行末でのパターンマッチ

検索キーの正規表現には行頭や行末の位置指定ができる。すなわち、検索キーの先頭に「^」を記述するとテキストの行の始め (行頭) にあるかどうか、検索キーの末尾に「\$」を記述するとテキストの行の末尾 (行末) にあるかどうかを検査できる。

例. 行頭でのマッチング検査

```
>>> txt = 'abcdabdefghefgh'  Enter    ←'abc' を 2 個含むテキスト
>>> ptn = r'abc'  Enter    ←検索キー 'abc' (位置指定なし)
>>> res = re.finditer( ptn, txt )  Enter    ←検索の実行
>>> for i in res:  Enter    ←検索結果を全て表示する処理
...     print( i.span() )  Enter
...  Enter
(0, 3)    ← 1 つめの一致
(4, 7)    ← 2 つめの一致
>>> ptn = r'^abc'  Enter    ←'^' を含む検索キー (行頭のための位置指定)
>>> res = re.finditer( ptn, txt )  Enter    ←検索の実行
>>> for i in res:  Enter    ←検索結果を全て表示する処理
...     print( i.span() )  Enter
...  Enter
(0, 3)    ←行頭の 1 つだけ一致
```

この例は、テキスト 'abcdabdefghefgh' の中のパターン 'abc' を探す例である。行末や行頭の位置指定をせずに実行すると 2 箇所 'abc' がみついているが、位置を指定すると 1 箇所のみパターンが見ついている。

次に、行末でのマッチングの例を示す。

例. 行末でのマッチング検査（先の例の続き）

```
>>> ptn = r'fgh'      Enter      ←検索キー 'fgh'（位置指定なし）
>>> res = re.finditer( ptn, txt )  Enter      ←検索の実行
>>> for i in res:      Enter      ←検索結果を全て表示する処理
...     print( i.span() )  Enter
...     Enter
(9, 12)                ← 1 つめの一致
(13, 16)               ← 2 つめの一致
>>> ptn = r'fgh$'      Enter      ← '$' を含む検索キー（行末のみの位置指定）
>>> res = re.finditer( ptn, txt )  Enter      ←検索の実行
>>> for i in res:      Enter      ←検索結果を全て表示する処理
...     print( i.span() )  Enter
...     Enter
(13, 16)               ←行末の 1 つだけ一致
```

4.3.2 置換処理： re.sub

正規表現のパターンにマッチする文字列を別の文字列に置き換える（置換処理）には sub メソッドを使用する。

例. 数字列の部分を「数字」に置換する：その 1

```
>>> tx1 = 'abc123def456'  Enter      ←数字を含む文字列
>>> re.sub(r'¥d+', '数字', tx1)  Enter      ←数字の部分を「数字」に置換
'abc 数字 def 数字'      ←置換結果
```

このように sub メソッドは置換結果の文字列を返す。

sub メソッドとは別の subn メソッドもある。これは置換処理に加えて、置換処理の対象となった部分の個数も取得する。

例. 数字列の部分を「数字」に置換する：その 2（先の例の続き）

```
>>> re.subn(r'¥d+', '数字', tx1)  Enter      ←数字の部分を「数字」に置換
('abc 数字 def 数字', 2)          ←置換結果と置換個数のタプル
```

4.3.2.1 複数行に渡る置換処理

複数行に渡る文字列（テキスト）の各行を置換処理する場合について考える。次のように各行の先頭に空白文字を持つテキストを想定する。

例. 行頭に空白文字を持つテキスト

```
>>> tx1 = '    1 行目¥n    2 行目¥n    3 行目'  Enter      ← 3 行に渡るテキスト
>>> print( tx1 )  Enter      ←内容確認
1 行目                ←テキストが
2 行目                ← 3 行にわたる
3 行目                ←文字列になっている
```

このようなテキスト tx1 の各行の先頭にある空白文字列を除去する試みを次に示す。

例. 行頭の '¥d+' にマッチする部分を空白文字に置き換える試み

```
>>> tx2 = re.sub( r'^¥s+', '', tx1 )  Enter      ← '^' を応用した行頭の置換処理
>>> print( tx2 )  Enter      ←内容確認
1 行目                ←最初の行のみ置換処理されている
2 行目                ←置換処理されていない
3 行目                ←置換処理されていない
```

この例のように、行頭でのパターンマッチが最初の行にのみ有効になっている。これは、tx1 のテキストの内容が複数の行に渡っているものの、文字列データとしては 1 行であることが原因である。

改行コードを含むテキストを複数の行から成るものと見なして、各行を別々に扱って置換処理するには `sub` メソッドにキーワード引数 `'flags=re.MULTILINE'` を与える。

例. テキストの各行を別々に扱う形での置換処理

```
>>> tx2 = re.sub( r'^s+', '', tx1, flags=re.MULTILINE )  ←行頭の置換処理
>>> print( tx2 )  ←内容確認
1 行目          ←置換処理されている
2 行目          ←置換処理されている
3 行目          ←置換処理されている
```

4.3.2.2 パターンマッチのグループを参照した置換処理

`sub` メソッドに与える正規表現にも括弧 `'(～)'` で括ったグループの記述が応用できる。

例. マッチしたグループを参照した置換処理

```
>>> tx1 = '6638137 兵庫県西宮市池開町'  ←郵便番号と地域
>>> re.sub( r'(\d{3})(\d{4})(.*)', r'\1-\2 \3', tx1 )  ←グループを用いた置換処理
'663-8137 兵庫県西宮市池開町'          ←置換結果
```

この例では `'6638137 兵庫県西宮市池開町'` という文字列の中からパターンマッチでいくつかのグループを取り出し、それら（「`\1`」に番号を付けて参照）を用いて置換処理を行っている。

第1のグループとして `'(\d{3})'` を記述しており「数字3文字」のパターンでマッチさせる。同様に第2のグループとして `'(\d{4})'` を記述しており「数字4文字」のパターンをマッチさせる。残りの部分（第3のグループ）は `'(.*)'` と記述して任意の文字列をマッチさせる。これらグループは「`\1`」に番号を付けて参照することができ、`sub` メソッドの第2引数のパターンの中の記述に使用されている。

4.3.3 文字列の分解への応用： `re.split`

`re.split` を用いると、正規表現のパターンにマッチする部分を区切りにして文字列を分解することができる。

例. 数字列を区切りにして文字列を分解

```
>>> tx1 = 'abc123def456ghi'  ←数字列で区切られた文字列
>>> re.split( r'\d+', tx1 )  ←正規表現を用いた分割処理
['abc', 'def', 'ghi']      ←分解結果（リストで得られる）
```

4.4 マルチスレッドとマルチプロセス

4.4.1 マルチスレッド

Python ではスレッド (thread) の考え方に基いて複数のプログラムを同時に並行して実行する¹⁹¹ ことができる。具体的には、threading モジュールを使用して関数やメソッドをメインプログラムから独立したスレッドとして実行する。このモジュールを使用するには次のようにして読み込む。

```
import threading
```

《 スレッド管理のための基本的なメソッド 》

● スレッドの生成 (コンストラクタ)

`threading.Thread(target=関数 (メソッド) の名前, args=(引数の並びのタプル), kwargs=キーワード引数の辞書)`

このメソッドの実行によりスレッドが生成され、それがスレッドオブジェクト (Thread クラス) として返される。

● スレッドの実行

`スレッドオブジェクト.start()`

start メソッドの実行により、スレッドの実行が開始する。この後、スレッドオブジェクトの `ident` 属性からスレッド ID (整数値) が得られる。

● スレッドの終了の同期

`スレッドオブジェクト.join()`

スレッドが終了するまで待つ。

2つの関数を別々のスレッドとして実行するプログラム `test12-1.py` を示す。

プログラム: `test12-1.py`

```
1 # coding: utf-8
2 import threading
3 import time
4
5 def th_1(name):      # 第1スレッド
6     for i in range(8):
7         print(name)
8         time.sleep(1)
9
10 def th_2(name):      # 第2スレッド
11     for i in range(3):
12         print('\t',name)
13         time.sleep(2)
14
15 # スレッドの生成
16 t1 = threading.Thread(
17     target=th_1, args=('Thread-1',))
18 t2 = threading.Thread(
19     target=th_2, args=('Thread-2',))
20 # スレッドの開始
21 t1.start();          t2.start()
22
23 # スレッドの終了待ち
24 t2.join(); print('\tThread-2 ended')
25 t1.join(); print('Thread-1 ended')
```

例. スクリプト `test12-1.py` の実行結果

```
Thread-1
          Thread-2
Thread-1
          Thread-2
Thread-1
Thread-1
          Thread-2
Thread-1
Thread-1
          Thread-2 ended
Thread-1
Thread-1
Thread-1 ended
```

2つのスレッドが同時に独立して動作していることがわかる。

解説

このプログラムでは関数 `th_1` と `th_2` を別々のスレッドとして実行する。16~19行目で引数を付けてスレッドを生成している。今回の例では引数の個数は1つなので、これをタプルとして渡すために、スレッドとなる関数に渡す引数の列の最後にコンマ `,` を付けている。21行目でスレッドを開始し、24,25行目でそれらの終了を待機している。

¹⁹¹マルチスレッドプログラミングにおいては、実際にはプログラムは「同時に並行」して実行されず、Python インタプリタの実行時間を各スレッドに分割配分する。後述のマルチプロセスプログラミングでは、CPU やコアの数によって「同時に並行」して実行されることがある。

■ スレッドにキーワード引数を渡す例

Thread オブジェクト生成時に 'target=' で指定した関数にキーワード引数を渡す場合はそれらを辞書の形で 'kwargs=' に渡す。

例. キーワード引数を取る関数の定義

```
>>> def job(x,**kwa): Enter ←キーワード引数を取る関数 job
...     print(' 第1引数:',x) Enter
...     print(' キーワード引数:',kwa) Enter
... Enter ←関数定義の記述の終了
>>> job(3,a=11,b=12,c=13) Enter ←関数 job の実行
第1引数: 3 ←出力
キーワード引数: {'a':11, 'b':12, 'c':13} ←キーワード引数の値
```

この関数をスレッドで実行するには次のようにする。

例. スレッドの関数にキーワード引数を渡す

```
>>> import threading Enter ↓スレッド生成
>>> th = threading.Thread(target=job,args=(3,),kwargs={'a':11,'b':12,'c':13}) Enter
>>> th.start() Enter ←スレッドを実行
第1引数: 3 ←出力
キーワード引数: {'a':11, 'b':12, 'c':13} ←キーワード引数の値
```

4.4.1.1 スレッドの実行状態の確認

スレッドが実行中かどうかを確認するには、当該スレッドオブジェクトに対して is_alive メソッドを実行する。このメソッドは、当該スレッドが実行中の場合は True を、そうでなければ False を返す。

is_alive メソッドでスレッドの終了を判定するサンプルプログラム test12-2.py を示す。

プログラム：test12-2.py

```
1  # coding: utf-8
2  import threading
3  import time
4
5  def th( n ):          # スレッドとして実行する関数
6      for i in range(n):
7          time.sleep(1)
8
9  # スレッドの生成
10 th_lst = []
11 for i in range(5):
12     t = threading.Thread(target=th, args=(i+1,))
13     th_lst.append( t )
14     t.start()
15
16 time.sleep(0.3)      # タイミング調整
17
18 while True:          # スレッド実行監視ループ
19     st = []
20     for t in th_lst:
21         st.append( t.is_alive() )
22     print('スレッド実行状態:',st)
23     time.sleep(1)
24     if not any(st):
25         break
```

解説.

プログラム中に定義されている関数 th はスレッドとして実行するもので、引数に与えた秒数が経過した後で終了する。11~14行目の for 文で、関数 th をスレッドとして5つ実行し、それらのスレッドオブジェクトのリストを th_lst に保持している。

このプログラムは join メソッドによってスレッドの終了を待機するのではなく、全てのスレッドの終了を定期的を確認することで処理を終了させる方法を取っている。18行目以降の部分は、全てのスレッドが終了しているかどうか

を繰り返し確認（1 秒間隔）するもので、全スレッドが終了していればプログラムを終了する。この際のスレッド状態の確認に `is.alive` メソッドを用いている。また、繰り返し処理の各回で5つのスレッドの実行状態のリストを `st` に作成（19～21 行目）し、`st` の全ての要素が `False` になった段階（24 行目）で `break` によって `while` の繰り返しを終了している。

このプログラムを実行した例を次に示す。

実行例.

```
スレッド実行状態: [True, True, True, True, True]
スレッド実行状態: [False, True, True, True, True]
スレッド実行状態: [False, False, True, True, True]
スレッド実行状態: [False, False, False, True, True]
スレッド実行状態: [False, False, False, False, True]
スレッド実行状態: [False, False, False, False, False]
```

スレッドの実行状態のリスト（変数 `st`）が1秒間隔で出力されることが確認できる。

4.4.1.2 実行中のスレッドのリストを取得する方法

`enumerate` 関数を使用すると、実行中のスレッドのリストが得られる。

書き方: `threading.enumerate()`

Thread オブジェクトのリストが得られる。ただし、この方法で得られるリストには Python のメインプログラムのスレッドも含まれている。

4.4.1.3 スレッド間の排他制御

`threading.Thread` オブジェクトによるスレッドとして関数を実行する際、その関数からの戻り値を直接的に受け取る方法はなく、スレッドからの情報を受取るにはグローバル変数を介するなど間接的な方法に頼ることになる。ここでは、複数のスレッドがグローバル変数を共有してアクセスする場合の注意点などについて解説する。

以下に示すサンプルプログラム `threadShare01-1.py` は、複数のスレッド（そのための関数）が1つのグローバル変数を共有してその値を更新する例である。（あまり良くない例）

プログラム: `threadShare01-1.py`（あまり良くない例）

```
1  # coding: utf-8
2  import threading
3  import time
4
5  shr = 0                # スレッドで共有する変数
6
7  def th(n):             # スレッドとして実行する関数
8      global shr
9      for m in range(10):
10         shr += 1        # 共有変数の値を1増加
11         time.sleep(0.01) # 少し待つ
12
13 # 10個のスレッドを生成
14 thList = [ threading.Thread(target=th,args=(n,)) for n in range(10) ]
15
16 for t in thList: t.start() # タイミングをずらしながら全スレッドを実行
17 for t in thList: t.join()  # 全スレッドの終了待ち
18 print('全スレッド終了. shr =',shr)
```

解説.

このプログラムでは、スレッドとして実行するための関数 `th` が定義されており、この関数はグローバル変数 `shr` の値を1増加させる処理を約1秒間に渡って10回実行する。この関数を10個のスレッドにして（14 行目）同時に実行（16 行目）し、全てのスレッドの終了を待機（17 行目）する。

このプログラムをスクリプトとして実行すると

全スレッド終了. shr = 100

と出力される。変数 `shr` の初期値は0であり、それを1増加させる処理を10回実行するスレッドを10個実行しているので当然の結果（ $10 \times 10 \rightarrow 100$ ）のように思われる。しかし、複数のプログラム（スレッドなど）が平行して実行

される状況においては、共有資源へのアクセスの衝突の問題を意識しなければならない。これに関して次のサンプルプログラム threadShare01-2.py を用いて考察する。

プログラム：threadShare01-2.py（アクセス衝突の例）

```
1  # coding: utf-8
2  import threading
3  import time
4
5  shr = 0          # スレッドで共有する変数
6
7  def th(n):       # スレッドとして実行する関数
8      global shr
9      for m in range(10):
10         tmp = shr          # 共有変数の値をローカル変数に読取り
11         time.sleep(0.007)  # 少し待つ
12         tmp += 1           # ローカル変数の値を1増加
13         shr = tmp          # ローカル変数の値を共有変数に上書き
14
15  # 10個のスレッドを生成
16  thList = [ threading.Thread(target=th,args=(n,)) for n in range(10) ]
17
18  for t in thList:   # タイミングをずらしながら全スレッドを実行
19      t.start()
20      time.sleep(0.01)
21
22  for t in thList: t.join()  # 全スレッドの終了待ち
23  print('全スレッド終了. shr =',shr)
```

このプログラムは先の threadShare01-1.py に似ているが、関数 th の定義が少し異なり、共有変数 shr の値をローカル変数 tmp に複製（10 行目）して少し待った（11 行目）後、tmp の値を 1 増加させて共有変数 shr に上書き（13 行目）している。このプログラムでは、1つのスレッドが time.sleep 関数で待っている間に他のスレッドが同様の処理をしており、変数 shr へのアクセスの衝突が起こる。

このプログラムをスクリプトとして実行すると

全スレッド終了. shr = 22

などと出力される。（処理環境によって shr の最終値が異なる）

■ ロックオブジェクト

Python のマルチスレッド処理では、**ロックオブジェクト**（lock オブジェクト）を用いることによって、スレッド間の共有資源へのアクセスにおける**排他制御**が実現できる。lock オブジェクトは Lock 関数で生成することができ、acquire, release を用いて排他制御の対象のプログラム範囲を指定する。

《ロックオブジェクトによる排他制御》

```
lock オブジェクト.acquire()
...
（排他制御するプログラムの記述）
...
lock オブジェクト.release()
```

lock オブジェクトは、互いに排他制御を行うスレッド間で同一のものを共用すること。

これに関して次のサンプルプログラム threadShare01-3.py を用いて解説する。

プログラム：threadShare01-3.py（排他制御の例）

```
1  # coding: utf-8
2  import threading
3  import time
4
5  L = threading.Lock()    # lockオブジェクトの生成
6  shr = 0                 # スレッドで共有する変数
7
8  def th(n):              # スレッドとして実行する関数
9      global shr
```

```

10     for m in range(10):
11         L.acquire()          # 排他制御範囲（ここから）
12         tmp = shr            # 共有変数の値をローカル変数に読取り
13         time.sleep(0.007)    # 少し待つ
14         tmp += 1             # ローカル変数の値を1増加
15         shr = tmp            # ローカル変数の値を共有変数に上書き
16         L.release()         # 排他制御範囲（ここまで）
17
18 # 10個のスレッドを生成
19 thList = [ threading.Thread(target=th,args=(n,)) for n in range(10) ]
20
21 for t in thList:          # タイミングをずらしながら全スレッドを実行
22     t.start()
23     time.sleep(0.01)
24
25 for t in thList: t.join()  # 全スレッドの終了待ち
26 print('全スレッド終了. shr =',shr)

```

このプログラムでは5行目で lock オブジェクト L を生成している。11行目に acquire メソッドを、16行目に release メソッドを記述しており、この範囲のプログラムが排他制御の対象となる。すなわち、あるスレッドがこの範囲を実行しているときに他のスレッドが同じ部分を実行しようとする、先に実行しているスレッドが release の部分に到達するまで待たされる。この排他制御により、上のプログラムでは変数 shr へのアクセスの衝突が回避できる。

このプログラムをスクリプトとして実行すると

全スレッド終了. shr = 100

と出力される。

参考 排他制御は with 構文（後の「4.21 with 構文」（p.340）で解説する）にも対応しており、threadShare01-3.py の関数 th の定義を次のように簡略化することもできる。

記述例：

```

1 def th(n):                # スレッドとして実行する関数
2     global shr
3     for m in range(10):
4         with L:           # 排他制御の対象部分
5             tmp = shr     # 共有変数の値をローカル変数に読取り
6             time.sleep(0.007) # 少し待つ
7             tmp += 1      # ローカル変数の値を1増加
8             shr = tmp     # ローカル変数の値を共有変数に上書き

```

注意)

最初のサンプルプログラム threadShare01-1.py においては、共有変数 shr へのアクセスの記述は、その変数の値を1増加させる文1行のみであるため、CPython 実装においては GIL (Global Interpreter Lock)¹⁹² によって結果的にアクセスの衝突が起こらないことになる。しかし、複数のスレッドで計算機資源を共有する場合は常にアクセスの衝突を意識しなければならない。

参考)

先のサンプルプログラムでは、グローバル変数 shr をスレッド間で共有しているが、これは実際のプログラミングではあまり好ましくない。実際のプログラミングでは情報の共有をする場合はそれらを何らかのオブジェクトの形にして、それをスレッド用関数の引数に渡すといった工夫をするべきである。

本書に示したサンプルプログラムでは、解説を簡素にするため、例外処理を施していない。実際のプログラミングでは十全な例外処理を施すこと。

¹⁹²CPython 実装では複数のスレッドが同時に実行されていても、ある瞬間に実行されているプログラムのステップは1つのみであるという制約。これに関する詳細は本書では割愛する。

4.4.2 マルチプロセス

マルチプロセスプログラミングでは、プログラムの実行単位（関数の実行など）を、独立したプロセスとして実行する。このため、複数の CPU やコアを搭載した計算機環境においては、OS の働きによって各プロセスの実行が複数の CPU やコアに適切に割り当てられる。Python には、マルチプロセスプログラミングのためのモジュール `multiprocessing` が標準的に提供されている。このモジュールを使用するには次のようにして Python 言語処理系に読み込む。

```
import multiprocessing
```

`multiprocessing` ライブラリが提供する API は `threading` ライブラリのものと使用方法がよく似ており、それらと対比しながら理解すると良い。

《 プロセス管理のための基本的なメソッド 》

● プロセスの生成（コンストラクタ）

`multiprocessing.Process(target=関数（メソッド）の名前, args=(引数の並びのタプル), kwargs=キーワード引数の辞書)`

このメソッドの実行によりプロセスが生成され、それがプロセスオブジェクト（`Process` クラス）として返される。

● プロセスの実行

`プロセスオブジェクト.start()`

`start` メソッドの実行により、プロセスの実行が開始する。この後、当該プロセスオブジェクトの `pid` 属性からプロセス ID（PID：整数値）が得られる。

● プロセス終了の同期

`プロセスオブジェクト.join()`

プロセスが終了するまで待つ。

2つの関数を別々のプロセスとして実行するプログラム `test12-1mp.py` を示す。

プログラム：`test12-1mp.py`

```
1 # coding: utf-8
2 import multiprocessing
3 import time
4
5 def pr_1(name):      # 第1プロセス用関数
6     for i in range(8):
7         print(name)
8         time.sleep(1)
9
10 def pr_2(name):      # 第2プロセス用関数
11     for i in range(3):
12         print('\t',name)
13         time.sleep(2)
14
15 if __name__ == '__main__':
16     # プロセスの生成
17     p1 = multiprocessing.Process(
18         target=pr_1, args=('Process-1',))
19     p2 = multiprocessing.Process(
20         target=pr_2, args=('Process-2',))
21     # プロセスの開始
22     p1.start();      p2.start()
23     # スレッドの終了待ち
24     p2.join();      print('\tProcess-2 ended')
25     p1.join();      print('Process-1 ended')
```

例。スクリプト `test12-1.py` の実行結果

```
Process-1
        Process-2
Process-1
Process-1
        Process-2
Process-1
Process-1
        Process-2
Process-1
Process-1
        Process-2 ended
Process-1
Process-1 ended
```

2つのプロセスが同時に独立して動作していることがわかる。

解説

このプログラムでは関数 `pr_1` と `pr_2` を別々のプロセスとして実行する。17～20行目で引数を付けてプロセスを生成している。今回の例では引数の個数は1つなので、これをタプルとして渡すために、プロセスとなる関数に渡す引数の列の最後にコンマ `,` を付けている。22行目でプロセスを開始し、24,25行目でそれらの終了を待機している。

4.4.2.1 プロセスごとに実行されるモジュール

各プロセスはそれぞれ独自の CPU リソースと仮想記憶空間を持つので、プロセスオブジェクトが start メソッドで実行を開始する際、そのプロセスのリソースとしてスクリプト全体を（モジュールとして）再度読み込む。このことは、複数のプロセスを生成して起動する際に問題を起こすことがある。例えば、起動された各々のプロセス内では当該スクリプトが再度読み込まれて実行されるので、各種オブジェクトを生成する処理の記述があると、それらの処理が各々のプロセス内で再度（不必要に）実行されてしまう。特に、スクリプト内にプロセスオブジェクトの生成と実行の処理が記述されていると、それらが、プロセス起動時に再帰的に（無限に）実行されることになる。

先のプログラム test12-1mp.py では、プロセスの生成と実行の処理を行う部分を

```
if __name__ == '__main__':
```

以下に記述している。これにより、プロセスオブジェクトの生成と実行が再帰的に（無限に）実行されることを防いでいる。すなわち、スクリプト test12-1mp.py が最初に Python 言語処理系によって実行されたときは、そのスクリプトのモジュール名は'__main__' であるのでこの if 文内のスイートが実行されるが、各プロセスの起動時に test12-1mp.py が再度読み込まれた際はモジュール名が別の名前¹⁹³ になるので、if 文のスイートは実行されず、プロセス生成と実行が再帰的に起こることはない。

モジュールの概念と、実行されるスクリプトのモジュール名については後の「4.8 モジュール、パッケージの作成による分割プログラミング」(p.299) で詳しく解説する。

4.4.2.2 プロセスの実行状態の確認

プロセスの実行状態の確認も threading の場合と同様に is_alive メソッドで行う。

プログラム：test12-2mp.py

```
1  # coding: utf-8
2  import multiprocessing
3  import time
4
5  def pr( n ):          # プロセスとして実行する関数
6      for i in range(n):
7          time.sleep(1)
8
9  if __name__ == '__main__':
10     # プロセスの生成と開始
11     pr_lst = []
12     for i in range(5):
13         p = multiprocessing.Process(target=pr, args=(i+1,))
14         pr_lst.append( p )
15         p.start()
16
17     time.sleep(0.3)      # タイミング調整
18     while True:         # プロセス実行監視ループ
19         st = []
20         for p in pr_lst:
21             st.append( p.is_alive() )
22         print('プロセス実行状態:',st)
23         time.sleep(1)
24         if not any(st):
25             break
```

解説.

プログラム中に定義されている関数 pr はプロセスとして実行するもので、引数に与えた秒数が経過した後で終了する。12～15 行目の for 文で、関数 pr をプロセスとして5つ実行し、それらのプロセスオブジェクトのリストを pr_lst に保持している。

このプログラムは join メソッドによってプロセスの終了を待機するのではなく、全てのプロセスの終了を定期的に確認することで処理を終了させる方法を取っている。18 行目以降の部分は、全てのプロセスが終了しているかどうかを繰り返し確認（1 秒間隔）するもので、全プロセスが終了していればプログラムを終了する。この際のプロセス状態の確認に is_alive メソッドを用いている。また、繰り返し処理の各回で5つのプロセスの実行状態のリストを st に

¹⁹³CPython では __mp_main__ となる。

作成（19～21 行目）し、st の全ての要素が False になった段階（24 行目）で break によって while の繰り返しを終了している。

このプログラムを実行した際の出力例を次に示す。

例. test12-2mp.py 実行の際のコンソール出力

```
プロセス実行状態: [True, True, True, True, True]
プロセス実行状態: [False, True, True, True, True]
プロセス実行状態: [False, False, True, True, True]
プロセス実行状態: [False, False, False, True, True]
プロセス実行状態: [False, False, False, False, True]
プロセス実行状態: [False, False, False, False, False]
```

4.4.2.3 実行中プロセスのリストを取得する方法

active_children 関数を使用すると、実行中のプロセスのリストが得られる。

書き方: multiprocessing.active_children()

Process オブジェクトのリストが得られる。ただし、この方法で得られるリストにはメインプログラムのプロセスは含まれない。

4.4.2.4 プロセスの強制終了

Process オブジェクトに terminate メソッドを実行することで、そのプロセスの実行を強制終了することができる。

書き方: Process オブジェクト.terminate()

「Process オブジェクト」のプロセスが実行中の場合はそれを強制終了する。実行中でない場合は何もしない。terminate メソッドは値を返さない (None)。

terminate メソッドでプロセスを強制終了する例を ProcTerminate01.py に示す。

プログラム: ProcTerminate01.py

```
1 # coding: utf-8
2 import multiprocessing
3 import time
4
5 def pr( ):          # プロセスとして実行する関数
6     while True:     # 際限なく実行される処理
7         time.sleep(1)
8
9 if __name__ == '__main__':
10     # プロセスの生成と開始
11     pr_lst = []
12     for i in range(5):
13         p = multiprocessing.Process(target=pr, args=())
14         pr_lst.append( p )
15         p.start()
16     time.sleep(0.3)    # タイミング調整
17
18     st = [p.is_alive() for p in pr_lst]
19     print('プロセス実行状態:',st)
20
21     while True:      # プロセス実行監視ループ
22         try:
23             # 終了対象プロセスのインデックスを入力
24             n = int( input('number to kill> ') )
25         except ValueError: # 入力が数値でなければ
26             n = 5         # 5とする
27         if 0 <= n <= 4:
28             pr_lst[n].terminate()    # プロセスを終了
29             pr_lst[n].join()         # 終了の待機
30         st = [p.is_alive() for p in pr_lst]
31         print('プロセス実行状態:',st)
32         if not any(st):              # 全てFalseなら
33             break                    # このプログラムを終了
```

このプログラムでは、プロセスとして実行する関数 pr を定義しており、この関数は自発的には終了しない（無限ループ）。この関数を5つのプロセスとして同時に実行し、それらをリスト pr_lst にして管理する。また、プロセス

の実行状態をリスト `st` にして表示する。キーボードから入力したインデックスのプロセスを `terminate` メソッドで終了して、全てのプロセスが終了した時点で当該プログラムは終了する。

このプログラムを実行した際のコンソールの出力例を次に示す。

例. `ProcTerminate01.py` 実行の様子

プロセス実行状態: [True, True, True, True, True]
number to kill> 4
プロセス実行状態: [True, True, True, True, False]
number to kill> 0
プロセス実行状態: [False, True, True, True, False]
number to kill> 3
プロセス実行状態: [False, True, True, False, False]
number to kill> 1
プロセス実行状態: [False, False, True, False, False]
number to kill> 2
プロセス実行状態: [False, False, False, False, False]

← 5つのプロセスが実行中
← インデックス 4 のプロセスを終了
← 4つのプロセスが実行中
← インデックス 0 のプロセスを終了
← 3つのプロセスが実行中
← インデックス 3 のプロセスを終了
← 2つのプロセスが実行中
← インデックス 1 のプロセスを終了
← 1つのプロセスが実行中
← インデックス 2 のプロセスを終了
← 全てのプロセスが終了

4.4.2.5 共有メモリと排他制御

各プロセスはそれぞれ独自の CPU リソースと仮想記憶空間を持つので、プロセス間での情報共有には特別な方法が必要となる。ここでは `multiprocessing` ライブラリが提供する共有メモリのための機能について解説する。

■ Value オブジェクトによる共有メモリ

Value オブジェクトは、1つの値をプロセス間で共有するためのものであり、作成には次のよう記述する。

書き方: `Value(型コード, 初期値)`

「型コード」で指定された型のデータを持つ Value オブジェクトを作成して返す。その値には `value` 属性としてアクセスする。Value オブジェクトは作成時に「初期値」を設定することができる。

「型コード」は表 36 に挙げる文字列である。

表 36: 型コード		
整数		
コード	解 説	C 言語での型
'b'	符号付き 8 ビット整数	char
'B'	符号なし 8 ビット整数	unsigned char
'h'	符号付き 16 ビット整数	short
'H'	符号なし 16 ビット整数	unsigned short
'i'	符号付き 32 ビット整数	int
'I'	符号なし 32 ビット整数	unsigned int
'l'	符号付き 32 ビット長整数	long
'L'	符号なし 32 ビット長整数	unsigned long
'q'	符号付き 64 ビット長長整数	long long
'Q'	符号なし 64 ビット長長整数	unsigned long long
浮動小数点数		
コード	解 説	C 言語での型
'f'	単精度浮動小数点数 (32 ビット)	float
'd'	倍精度浮動小数点数 (64 ビット)	double
その他		
コード	解 説	C 言語での型
'c'	ASCII の 1 文字 (例. b'a')	char

Value オブジェクトはプロセス用関数の引数として渡して共有するのが一般的である。

■ ロックオブジェクト

threading ライブラリの場合と同様 (p.258) に、**ロックオブジェクト** (lock オブジェクト) によってプロセス間での共有メモリの**排他制御**ができる。ロックオブジェクトは、互いに排他制御を行うプロセスの間で同一のものを共用すること。

先に示したマルチスレッドの排他制御のサンプルプログラム threadShare01-3.py (p.258) に倣い、マルチプロセスで同様の処理を実現した procShare01.py 示す。

プログラム：procShare01.py

```
1  # coding: utf-8
2  import multiprocessing
3  import time
4
5  def pr(n, shr, L):          # プロセスとして実行する関数
6      for m in range(10):
7          L.acquire()        # 排他制御範囲 (ここから)
8          tmp = shr.value     # 共有変数の値をローカル変数に読取り
9          time.sleep(0.007)   # 少し待つ
10         tmp += 1            # ローカル変数の値を1増加
11         shr.value = tmp     # ローカル変数の値を共有変数に上書き
12         L.release()        # 排他制御範囲 (ここまで)
13
14  if __name__ == '__main__':
15      L = multiprocessing.Lock()          # Lockオブジェクトの生成
16      shr = multiprocessing.Value('i', 0) # プロセス間で共有するオブジェクト
17      # 10個のプロセスを生成
18      prList = [multiprocessing.Process(target=pr, args=(n, shr, L))
19                  for n in range(10)]
20      # タイミングをずらしながら全プロセスを実行
21      for p in prList:
22          p.start()
23          time.sleep(0.01)
24      # 全プロセスの終了待ち
25      for p in prList: p.join()
26      print('全プロセス終了. shr.value =', shr.value)
```

このプログラムでは、Value オブジェクト shr を全プロセスで共有しており、ロックオブジェクト L によってそのアクセスを排他制御している。これにより shr.value の値が正常に増加し、コンソールで実行すると

全プロセス終了. shr.value = 100

と出力される。

参考) 排他制御は with 構文 (後の「4.21 with 構文」(p.340) で解説する) にも対応しており、procShare01.py の関数 pr の定義を次のように簡略化することもできる。

記述例：

```
1  def pr(n, shr, L):          # プロセスとして実行する関数
2      for m in range(10):
3          with L:              # 排他制御範囲
4              tmp = shr.value  # 共有変数の値をローカル変数に読取り
5              time.sleep(0.007) # 少し待つ
6              tmp += 1         # ローカル変数の値を1増加
7              shr.value = tmp  # ローカル変数の値を共有変数に上書き
```

参考) ここで解説したもの以外にも multiprocessing ライブラリは多くのロック機能を提供している。詳しくは Python の公式インターネットサイトなどを参照のこと。

■ Array オブジェクト

データの配列をプロセス間で共有するには Array オブジェクトを用いる。作成には次のように記述する。

書き方： Array(型コード, 初期値)

「型コード」は Value オブジェクトの場合に準ずる。「初期値」はリストなどの形で与える。Array オブジェクトの要素へのアクセスには、リストの場合と同様の添字 (スライス) が使える。

1つの配列（Array オブジェクト）を複数のプロセスで共有してアクセスする例を procShare02.py に示す。

プログラム：procShare02.py

```
1 # coding: utf-8
2 import multiprocessing
3 import time
4
5 def pr(n, shr, L): # プロセスとして実行する関数
6     time.sleep(1) # 1秒待機
7     L.acquire() # 排他制御範囲（ここから）
8     shr[n] = n # 共有メモリ（配列）に値を設定
9     L.release() # 排他制御範囲（ここまで）
10
11 if __name__ == '__main__':
12     L = multiprocessing.Lock() # Lockオブジェクトの生成
13     shr = multiprocessing.Array('i', [-1]*5) # プロセス間で共有する配列
14     print('初期状態:', list(shr))
15     # 5個のプロセスを生成
16     prList = [multiprocessing.Process(target=pr, args=(n, shr, L))
17               for n in range(5)]
18     # 全プロセスを実行
19     for p in prList: p.start()
20     # 全プロセスの終了待ち
21     for p in prList: p.join()
22     print('最終状態:', list(shr))
```

このプログラムでは、Array オブジェクト shr を複数のプロセスで共有し、各プロセスがその要素の値を書き換える。初期状態では shr の全要素は -1 であり、全てのプロセスが終了すると、各プロセスの起動順番 n に対応するインデックスの要素の値が n となる。

例. コンソールで procShare02.py を実行した際の出力

初期状態: [-1, -1, -1, -1, -1]
最終状態: [0, 1, 2, 3, 4]

■ Manager

Manager を用いると、リストや辞書などを共有メモリとして使用することができる。具体的には、**マルチプロセッシングマネージャオブジェクト**（SyncManager クラスのオブジェクトで、単に**マネージャ**と呼ぶことも多い）を作成し、その下で共有リストや共有辞書を作成する。

書き方: Manager()

この記述がマネージャを作成して返す。このようにして得られたマネージャで共有リスト、共有辞書を作成する。

共有リスト: マネージャ.list()

共有辞書: マネージャ.dict()

これらの記述で生成されたリストや辞書を、プロセス用の関数に引数として渡す。

共有リストを2つのプロセスで共有して更新する例を procShare03list.py に示す。

プログラム：procShare03list.py

```
1 # coding: utf-8
2 from multiprocessing import Process, Lock, Manager # 必要機能を個別に読み込む
3 import time
4
5 def pr( lst, e, L ): # プロセス用関数
6     time.sleep(1) # 少し待つ
7     L.acquire() # 排他制御（ここから）
8     lst.append(e) # 共有メモリのリストの処理
9     L.release() # 排他制御（ここまで）
10
11 if __name__ == '__main__':
12     L = Lock() # 排他制御のためのロックオブジェクト
13     M = Manager() # 共有メモリを作成するManager
14     lst = M.list() # 共有メモリとしてのリスト
15     p1 = Process(target=pr, args=(lst, 'a', L)) # 1番目のプロセス
16     p2 = Process(target=pr, args=(lst, 'b', L)) # 2番目のプロセス
17     p1.start(); p2.start() # 2つのプロセスを起動
18     p1.join(); p2.join() # 2つのプロセスの終了を待機
```

```
19 print('処理結果:',lst)
```

このプログラムでは、共有リスト lst を2つのプロセス p1, p2 で共有して要素を追加する。このプログラムを実行すると「処理結果: ['a', 'b']」と出力される。p1, p2 の実行のタイミングによっては「処理結果: ['b', 'a']」と出力されることもある。

共有辞書を2つのプロセスで共有して更新する例を procShare03dict.py に示す。

プログラム：procShare03dict.py

```
1 # coding: utf-8
2 from multiprocessing import Process, Lock, Manager # 必要機能を個別に読み込む
3 import time
4
5 def pr( d, k, v, L ): # プロセス用関数
6     time.sleep(1) # 少し待つ
7     L.acquire() # 排他制御 (ここから)
8     d[k] = v # 共有メモリの辞書の処理
9     L.release() # 排他制御 (ここまで)
10
11 if __name__ == '__main__':
12     L = Lock() # 排他制御のためのロックオブジェクト
13     M = Manager() # 共有メモリを作成するManager
14     d = M.dict() # 共有メモリとしての辞書
15     p1 = Process(target=pr, args=(d, 'apple', 'りんご', L)) # 1番目のプロセス
16     p2 = Process(target=pr, args=(d, 'orange', 'みかん', L)) # 2番目のプロセス
17     p1.start(); p2.start() # 2つのプロセスを起動
18     p1.join(); p2.join() # 2つのプロセスの終了を待機
19     print('処理結果:',d)
```

このプログラムでは、共有辞書 d を2つのプロセス p1, p2 で共有してエントリを追加する。このプログラムを実行すると「処理結果: {'apple': 'りんご', 'orange': 'みかん'}」と出力される。p1, p2 の実行のタイミングによっては「処理結果: {'orange': 'みかん', 'apple': 'りんご'}」と出力されることもある。

【Manager の名前空間】

Manager が提供する名前空間 (Namespace) を使用すると、リストや辞書に限らず様々な型のデータをプロセス間で共有することができる。

書き方： マネジャ.Namespace()

「マネジャ」の下で名前を管理する名前空間プロキシオブジェクト (NamespaceProxy クラス) を生成して返す。

名前空間プロキシではドット表記の属性管理ができる。

書き方： 名前空間プロキシ.属性名

この形式で値の代入や参照ができる。

名前空間プロキシを2つのプロセスで共有する例を procShare03ns.py に示す。

プログラム：procShare03ns.py

```
1 # coding: utf-8
2 from multiprocessing import Process, Lock, Manager # 必要機能を個別に読み込む
3 import time
4
5 def pr1( ns, L ): # プロセス用関数1
6     time.sleep(1) # 少し待つ
7     L.acquire() # 排他制御 (ここから)
8     ns.i = 123; ns.f = 3.14159; ns.s = '文字列'; ns.b = True
9     L.release() # 排他制御 (ここまで)
10
11 def pr2( ns, L ): # プロセス用関数2
12     time.sleep(1) # 少し待つ
13     L.acquire() # 排他制御 (ここから)
14     ns.lst = ['a','b']; ns.tpl = ('c','d');
15     ns.d = {'apple':'りんご'}; ns.st = {7,11,13}
16     L.release() # 排他制御 (ここまで)
17
18 if __name__ == '__main__':
19     L = Lock() # 排他制御のためのロックオブジェクト
```

```

20 M = Manager()          # 共有メモリを作成するManager
21 N = M.Namespace()     # 共有メモリ用の名前空間
22 p1 = Process(target=pr1, args=(N, L))      # 1番目のプロセス
23 p2 = Process(target=pr2, args=(N, L))      # 2番目のプロセス
24 p1.start();           p2.start()           # 2つのプロセスを起動
25 p1.join();            p2.join()            # 2つのプロセスの終了を待機
26 print('処理結果:')
27 print(f'  N.i: {N.i},\tN.f: {N.f},\tN.s: {N.s},\tN.b: {N.b}')
28 print(f'  N.lst: {N.lst},\tN.tpl: {N.tpl},')
29 print(f'  N.d: {N.d},\tN.st: {N.st}')

```

このプログラムでは、名前空間プロキシ N を2つのプロセス p1, p2 で共有し、各プロセスはそれの属性に対して値を設定する。このプログラムを実行した際の出力例を示す。

例. コンソールで procShare03ns.py を実行した際の出力

処理結果:

```

N.i: 123,    N.f: 3.14159,  N.s: 文字列,    N.b: True
N.lst: ['a', 'b'],      N.tpl: ('c', 'd'),
N.d: {'apple': 'りんご'}, N.st: {11, 13, 7}

```

様々な型のデータが共有できていることがわかる。

参考) ここで解説したもの以外にも Manager は多くの機能を提供している。詳しくは Python の公式インターネットサイトなどを参照のこと。

■ shared_memory

Python 3.8 の版から、multiprocessing ライブラリに shared_memory が提供されている。この中に定義されている SharedMemory クラスは、プロセス間で共有する汎用の記憶域に関するものであり、次のようにしてコンストラクタを呼び出すことで共有メモリのインスタンスが生成される。

書き方: SharedMemory(create=True, size=メモリサイズ)

指定した「メモリサイズ」(バイト単位の整数値) の共有メモリを生成して返す。

共有メモリ (SharedMemory のインスタンス) の name プロパティにはその名前が文字列の形で保持されており、子プロセス側ではこれを指定して既存の共有メモリを取得する。

書き方: SharedMemory(name=名前)

「名前」を持つ既存の共有メモリのインスタンスを返す。

共有メモリが持つ記憶域 (バッファ領域) は buf 属性としてアクセスできる。これは編集可能なバイト列であり、後の「4.13 編集可能なバイト列: bytearray」(p.317) で解説する bytearray 型のオブジェクトと同様の扱いが可能である。

共有メモリを使用している各プロセスにおいて、その使用を終了するには、close メソッドを実行する。

書き方: 共有メモリ.close()

また、共有メモリを廃棄 (解放) するには unlink メソッドを実行する。

書き方: 共有メモリ.unlink()

次に示すサンプル procShare04.py は、2つのプロセスが1つの共有メモリに値を書き込む処理の例である。

プログラム: procShare04.py

```

1  # coding: utf-8
2  from multiprocessing import Process, Lock          # 必要なクラス
3  from multiprocessing.shared_memory import SharedMemory # 共有メモリのクラス
4
5  # プロセス用関数1
6  def pr1( n, L ):
7      shr = SharedMemory( name=n )      # 名前を指定して共有メモリ名を取得
8      B = shr.buf                        # そのバッファ領域を取得
9      L.acquire()
10     B[0] = 0xe5;          B[1] = 0x85;          B[2] = 0xb1      # 「共」
11     L.release()
12     shr.close()          # 共有メモリへのアクセスを終了
13

```



```

14 # プロセス用関数2
15 def pr2( n, L ):
16     shr = SharedMemory( name=n )      # 名前を指定して共有メモリ名を取得
17     B = shr.buf                       # そのバッファ領域を取得
18     L.acquire()
19     B[3] = 0xe6;          B[4] = 0x9c;          B[5] = 0x89      # 「有」
20     L.release()
21     shr.close()      # 共有メモリへのアクセスを終了
22
23 if __name__ == '__main__':
24     shr = SharedMemory(create=True, size=6)      # 共有メモリの作成
25     L = Lock()      # 排他制御のためのロックオブジェクト
26     p1 = Process(target=pr1, args=(shr.name, L)) # プロセス1
27     p2 = Process(target=pr2, args=(shr.name, L)) # プロセス2
28     p1.start();      p2.start()      # 2つのプロセスを起動
29     p1.join();      p2.join()      # 2つのプロセスの終了を待機
30     s = bytes(shr.buf).decode('utf-8')      # 共有メモリの内容を文字列に変換して、
31     print(s)      # 表示する。
32     shr.close()      # 共有メモリへのアクセスを終了
33     shr.unlink()      # 共有メモリの領域の解放

```

このプログラムでは pr1, pr2 が子プロセスとして起動され、共有メモリ shr のバッファ領域に値を書き込む。子プロセスが終了すると、メインプログラム側で shr のバッファ領域の内容を文字列に変換して出力する。

pr1 内では共有メモリのバッファの前半に「共」という文字の文字コード (utf-8) を書き込み、pr2 内では共有メモリのバッファの後半に「有」という文字の文字コードを書き込む。その後、メインプログラムは共有メモリのバッファの内容を utf-8 エンコーディングで文字列に変換して出力する。

このプログラムを実行すると「共有」という文字が出力される。

【shared_memory の応用例】

SharedMemory オブジェクトは汎用のバイト領域を持つので、様々な用途に応用できる。ここでは NumPy ライブラリ¹⁹⁴ が提供する数値配列をプロセス間で共有する事例を挙げる。以下に示す procShare05.py は、3行 150列の2次元配列をプロセス間で共有し、三角関数の値を子プロセスで計算し、メインプログラムで計算結果をグラフにプロットする処理の例である。

プログラム：procShare05.py

```

1  # coding: utf-8
2  from multiprocessing import Process, Lock      # 必要なクラス
3  from multiprocessing.shared_memory import SharedMemory # 共有メモリのクラス
4  import numpy as np      # 数値演算ライブラリ
5  import matplotlib.pyplot as plt      # グラフ作成ライブラリ
6
7  # プロセス用関数1
8  def pSin( n, shape, type, L ):
9      shr = SharedMemory( name=n )      # 名前を指定して共有メモリ名を取得
10     # 共有メモリを元に NumPy 配列のビューを作る
11     shrA = np.ndarray(shape, dtype=type, buffer=shr.buf)
12     L.acquire()
13     shrA[1,:] = np.sin( shrA[0,:] )      # 正弦関数を算出
14     L.release()
15     shr.close()      # 共有メモリへのアクセスを終了
16
17 # プロセス用関数2
18 def pCos( n, shape, type, L ):
19     shr = SharedMemory( name=n )      # 名前を指定して共有メモリ名を取得
20     # 共有メモリを元に NumPy 配列のビューを作る
21     shrA = np.ndarray(shape, dtype=type, buffer=shr.buf)
22     L.acquire()
23     shrA[2,:] = np.cos( shrA[0,:] )      # 余弦関数を算出
24     L.release()
25     shr.close()      # 共有メモリへのアクセスを終了
26
27 if __name__ == '__main__':
28     dummy = np.zeros((3, 150))      # メモリサイズを調べるためのダミー配列の作成

```

¹⁹⁴利用に際しては事前にインストールする必要あり。ライブラリの詳細については公式インターネットサイト <https://numpy.org/> を参照のこと。拙書「Python3 ライブラリブック」でも基本的な使用方法を解説しています。

```

29 shr = SharedMemory(create=True, size=dummy.nbytes) # 共有メモリの作成
30 # 共有メモリのためのNumPy配列のビューを作る
31 shrA = np.ndarray(dummy.shape, dtype=dummy.dtype, buffer=shr.buf)
32 del dummy # ダミーの消去
33 # 配列の先頭行にx軸の値 ( $-2\pi \sim 2\pi$ ) を設定
34 shrA[0,:] = np.linspace(-2*np.pi, 2*np.pi, 150)
35 L = Lock() # 排他制御のためのロックオブジェクト
36 p1 = Process(target=pSin, args=(shr.name, shrA.shape, shrA.dtype, L))
37 p2 = Process(target=pCos, args=(shr.name, shrA.shape, shrA.dtype, L))
38 p1.start(); p2.start() # 実行
39 p1.join(); p2.join() # 終了の待機
40 # 作図処理
41 plt.figure(figsize=(6,4))
42 plt.plot(shrA[0,:], shrA[1,:], label='sin(x)')
43 plt.plot(shrA[0,:], shrA[2,:], label='cos(x)')
44 plt.xlabel('x'); plt.ylabel('y')
45 plt.title('Triangular Function: y=sin(x), y=cos(x)')
46 plt.grid(); plt.legend()
47 plt.show()
48 # 共有記憶の終了と廃棄
49 shr.close() # 共有メモリへのアクセスを終了
50 shr.unlink() # 共有メモリの領域の解放

```

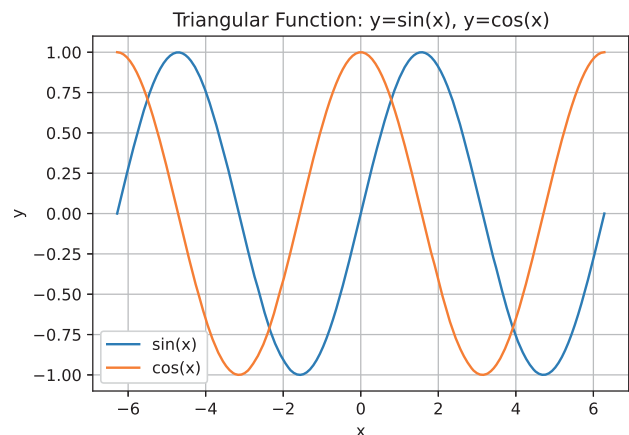
このプログラムのメイン部で共有メモリ shr を生成している。この際、3 行 150 列の ndarray (NumPy の数値配列) に相当するサイズのバッファ領域を確保するため、dummy 配列を作成してそのバイトサイズを nbytes 属性で取得している。次に、shr のバッファ領域 (shr.buf) を記憶域として持つ ndarray を shrA として作成 (ndarray コンストラクタの引数 'buffer=' に記憶域を与える) し、以後、メイン部ではこの shrA を数値配列として使用する。

このプログラムでは、正弦関数の値を pr1 (プロセス p1) で、余弦関数の値を pr2 (プロセス p2) で算出して共有メモリに書き込む。各プロセスには、共有メモリの名前が引数を介して与えられ、それを引用することで、メインプロセスが確保した共有メモリを取得している。また各プロセス内では、メインプロセスと同様の方法で、共有メモリから ndarray を作成している。

子プロセスの処理が全て終了して計算結果として完成した shrA の内容を、メイン部で matplotlib ライブラリ¹⁹⁵ を使用 (41~47 行目) してグラフとしてプロットしている。

このプログラムを実行すると右のようなグラフが表示される。

- plt.figure でグラフサイズを指定している。
- plt.plot でグラフを描画している。
- plt.xlabel, plt.ylabel で軸ラベルを書いている。
- plt.title でグラフタイトルを上部に書いている。
- plt.grid で格子を描いている。
- plt.legend で凡例を作成している。
- plt.show で最終的にグラフを表示している。



procShare05.py の実行によって作成されたグラフ

¹⁹⁵ 利用に際しては事前にインストールする必要あり。ライブラリの詳細については公式インターネットサイト <https://matplotlib.org/> を参照のこと。拙書「Python3 ライブラリブック」でも基本的な使用方法を解説しています。

4.4.3 concurrent.futures

Python 3.2 の版から並行プログラミングのためのライブラリ `concurrent.futures` が標準的に提供されている。このライブラリは、`threading`、`multiprocessing` ライブラリを基に構築されており、高機能な API を提供する。

このモジュール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 concurrent import futures
```

注意) `concurrent.futures` モジュールを用いたマルチプロセスの実行は、モジュール名 `__main__` の下で行うものとする。これは `multiprocessing` ライブラリの場合と同じ理由による。

4.4.3.1 ProcessPoolExecutor

`concurrent.futures` には、マルチプロセスの形で関数を実行するための `ProcessPoolExecutor` というクラスが用意されており、このクラスのインスタンスが複数のプロセスの実行を管理する。具体的には、

```
futures.ProcessPoolExecutor(max_workers=プールの最大数)
```

と記述する。**プール**とは、一連のスレッドやプロセスを管理する作業の単位である。`ProcessPoolExecutor` の引数 `'max_workers='` には使用する**プロセス**の最大値を与える¹⁹⁶。これを省略すると当該環境の CPU の数が採用される。例えば、プールの最大数を 4（最大 4 個のプロセスを管理）としてインスタンスを生成する場合は次のように記述する。

```
exe = futures.ProcessPoolExecutor(max_workers=4)
```

この結果、`exe` というインスタンスが生成され、これに対して `submit` メソッドを実行することでプロセスを実行することができる。例えば、`ProcessPoolExecutor` オブジェクト `exe` によって関数 `f('arg')` を実行するには、次のように記述する。

```
exe.submit(f, 'arg')
```

これによって、関数 `f('arg')` が独立したプロセスとして実行される。複数の引数を取る関数を実行するには、`submit` の引数にコンマで区切ってそのまま引数を書き並べる。

`submit` メソッドは `Future` オブジェクトを返し、これによってプロセスの実行を管理する。

`submit` メソッドによって起動した全てのプロセスが終了して `ProcessPoolExecutor` オブジェクトが不要になった際は `shutdown` メソッド実行する。

例. `ProcessPoolExecutor` オブジェクト `exe` の終了処理

```
exe.shutdown()
```

`shutdown` メソッドを実行すると、実行中のプロセスが全て終了するまで待ち、プロセス実行に使用した計算機資源が解放される。

重要) `concurrent.futures` モジュールには、`ProcessPoolExecutor` とは別に `ThreadPoolExecutor` クラスも提供されており、`ProcessPoolExecutor` クラスとほぼ同じ使用方法でマルチスレッドを実現する。

注意) `ProcessPoolExecutor` では `multiprocessing` ライブラリの共有メモリの一部機能 (`Value`, `Array`, `SharedMemory`) や `Lock` は使えるが、`Manager` は使うことができない。`ProcessPoolExecutor` は `multiprocessing` の完全な上位互換ではないことに注意すること。

4.4.3.2 プロセスの実行状態と戻り値

プロセスの実行状態を調べるには、当該 `Future` オブジェクトに対して `running` メソッドを実行する。このメソッドは、プロセスが「実行中」の場合に `True` を、そうでない場合に `False` を返す。ただし、実際にプロセスの関数が実行を開始する直前にプロセス管理の準備の処理が始まり、この段階から「実行中」となる。また、プロセスの関数が実行を終えた直後も短時間ではあるがプロセス管理の後処理があり「実行中」の状態が続く。

終了したプロセスの戻り値は当該 `Future` オブジェクトに対して `result` メソッドを実行することで得られる。

マルチプロセス実行に関するサンプルプログラム `test12-3.py` を示す。

¹⁹⁶これに関しては後の「4.4.3.3 同時に実行されるプロセスの個数について」(p.272)で例を挙げて解説する。

プログラム：test12-3.py

```
1  # coding: utf-8
2  from concurrent import futures
3  import time
4
5  def prc( n ):          # プロセスとして実行する関数
6      t0 = time.time()
7      for i in range(n):
8          time.sleep(1)
9      return time.time() - t0
10
11 # プロセスの実行
12 if __name__ == '__main__':
13     exe = futures.ProcessPoolExecutor(max_workers=4)
14     prc_lst = []        # Futureオブジェクト管理用のリスト
15     for i in range(4):
16         f = exe.submit( prc, i+1 )
17         prc_lst.append( f )
18     # タイミング調整
19     time.sleep(0.3)
20     # プロセス実行監視ループ
21     while True:
22         st = []
23         for f in prc_lst:
24             st.append( f.running() )
25         print( 'プロセスの実行状態:', st )
26         time.sleep(1)
27         if not any(st):
28             break
29     exe.shutdown()
30     # プロセスの戻り値の出力
31     for f in prc_lst:
32         print( f, 'の戻り値:', f.result() )
```

解説.

プログラム中に定義されている関数 prc はプロセスとして実行するもので、引数に与えた秒数が経過した後、実際に実行に要した時間を返して終了する。15～17 行目の for 文で、関数 prc をプロセスとして 4 つ実行し、それらの Future オブジェクトのリストを prc_lst に保持している。

21～28 行目の部分は、全てのプロセスが終了しているかどうかを繰り返し確認（1 秒間隔）するもので、全プロセスが終了していれば各プロセスの戻り値を表示してプログラムを終了する。この際のスレッド状態の確認に running メソッドを用いている。また、while による繰り返し処理の各回で 4 つのプロセスの実行状態のリストを st に作成（22～24 行目）し、st の全ての要素が False になった段階（27 行目）で break によって while の繰り返しを終了する。その後、31～32 行目で各プロセスの戻り値を表示している。

このプログラムを実行した例を次に示す。

実行例.

```
プロセスの実行状態: [True, True, True, True]
プロセスの実行状態: [False, True, True, True]
プロセスの実行状態: [False, False, True, True]
プロセスの実行状態: [False, False, False, True]
プロセスの実行状態: [False, False, False, False]
<Future at 0x1ec590a2b80 state=finished returned float> の戻り値: 1.00010085105896
<Future at 0x1ec590b9b20 state=finished returned float> の戻り値: 2.005520820617676
<Future at 0x1ec590b9f70 state=finished returned float> の戻り値: 3.0089118480682373
<Future at 0x1ec590cb070 state=finished returned float> の戻り値: 4.012871980667114
```

注意)

ProcessPoolExecutor はプロセスに引数を渡す、あるいは戻り値を受け取る際に pickle モジュール¹⁹⁷ の機能を使用しているので、データの受け渡しにはこのことに由来する制約がある。

¹⁹⁷後の「4.10 オブジェクトの保存と読み込み：pickle モジュール」(p.306) で解説する。

4.4.3.3 同時に実行されるプロセスの個数について

ProcessPoolExecutor のインスタンスを生成する際に与える引数 `max_workers=n` は、同時に実行されるプロセスの個数の上限である。この値を超える個数のプロセスが `submit` メソッドによって投入された場合は、投入順に n 個のプロセスが実行され、それ以降のプロセスは実行待ちとなる。これについて、次のサンプルプログラム `test12-4.py` で確認する。

プログラム：test12-4.py

```
1  # coding: utf-8
2  from concurrent import futures
3  import time
4
5  def prc( n ):                                # プロセスとして実行する関数
6      pname = 'p'+str(n)
7      print( pname+'開始' )
8      time.sleep(0.5)
9      print( pname+'終了' )
10     return pname
11
12 # プロセスの実行
13 if __name__ == '__main__':
14     exe = futures.ProcessPoolExecutor(max_workers=1)    # 引数の値を変更して試す
15     prc_lst = []    # Futureオブジェクト管理用のリスト
16     for i in range(4):
17         f = exe.submit( prc, i+1 )
18         prc_lst.append( f )
19     # 全プロセス終了の同期
20     r = []
21     for f in futures.as_completed(prc_lst): r.append(f.result())
22     # 終了処理
23     print( '各プロセスの戻り値 :', r )
24     exe.shutdown()
```

解説.

このプログラムでは関数 `prc` をプロセスとして実行する。この関数は実行開始のメッセージを出力した後 0.5 秒待ち、終了メッセージを出力して処理を終える。16～18 行目の `for` 文で `prc` 関数をプロセスとして 4 個投入する。

このプログラムの `max_workers` (14 行目) を変えて実行した例を次に示す。

実行例. `max_workers=1` の場合

```
p1 開始
p1 終了
p2 開始
p2 終了
p3 開始
p3 終了
p4 開始
p4 終了
各プロセスの戻り値 : ['p1', 'p2', 'p3', 'p4']
```

同時に実行できるプロセス個数が 1 なので、各プロセスが 1 つずつ順番に開始、終了している様子が見える。

実行例. `max_workers=2` の場合

```
p1 開始
p2 開始
p1 終了
p2 終了
p3 開始
p4 開始
p3 終了
p4 終了
各プロセスの戻り値 : ['p1', 'p2', 'p3', 'p4']
```

同時に実行できるプロセス個数が 2 なので、プロセスが 2 つずつ同時に開始、終了している様子が見える。すなわち、`p1,p2` が同時に開始し、これらが終了するまで `p3,p4` は実行されずに待っている。`p1,p2` が終了すると、次の `p3,p4` が 2 つ同時に実行されている。

4.4.3.4 プロセス終了の同期

`as_completed` 関数を使用することでプロセスの終了を同期することができる。先のプログラム `test12-4.py` ではこの関数を用いて全てのプロセスの終了を待機している。

書き方： `as_completed(イテラブル)`

「イテラブル」は Future オブジェクトを要素とするもので、各要素が示すプロセスが終了する度に、その Future の結果を含んだ形のものを順次返すイテレータを作成する。

先のプログラム test12-4.py の prc_lst は、実行されるプロセスを示す Future オブジェクトを要素として持つリストであり、21 行目で as_completed 関数によって、終了した Future オブジェクトを順次返すイテレータを作成している。またそのイテレータを for 文によって処理しており、結果的に全てのプロセスが終了したときに for 文が終了する。

4.4.4 マルチスレッドとマルチプロセスの実行時間の比較

ここでは、サンプルプログラム mproc01.py を使って、マルチスレッドとマルチプロセスの間の実行時間の差異を調べる方法を例示する。プログラム中には関数 calcf が定義されており、これは円軌道のシミュレーションを実行する（円を表現する差分方程式の数値解を求める）もの¹⁹⁸である。これを各種の異なる方法で実行する。

プログラム：mproc01.py

```
1  # coding: utf-8
2  # 必要なモジュールの読み込み
3  import time
4  import threading
5  from concurrent import futures
6
7  # 対象の関数：円軌道の精密シミュレーション
8  def calcf(name):
9      dt = 0.0000001
10     x = 1.0; y = 0.0
11     t1 = time.time()
12     for i in range(62831853):
13         x -= y * dt; y += x * dt
14     t = time.time() - t1
15     print(name, '- time:', t)
16
17 #####
18 #   マルチスレッドとマルチプロセスによる実行時間の比較実験   #
19 #####
20 if __name__ == '__main__':
21
22     #   threadingモジュールによる実験
23     #   print('--- By threading ---')
24     #   p1 = threading.Thread(target=calcf, args=('p1',))
25     #   p2 = threading.Thread(target=calcf, args=('p2',))
26     #   p3 = threading.Thread(target=calcf, args=('p3',))
27     #   p1.start()
28     #   p2.start()
29     #   p3.start()
30     #   p1.join()
31     #   p2.join()
32     #   p3.join()
33
34     #   concurrent.futuresモジュールによるマルチプロセスの実験
35     #   print('--- By concurrent.futures : [ProcessPoolExecutor] ---')
36     #   exe = futures.ProcessPoolExecutor(max_workers=4)
37     #   f1 = exe.submit(calcf, 'p1')
38     #   f2 = exe.submit(calcf, 'p2')
39     #   f3 = exe.submit(calcf, 'p3')
40     #   f4 = exe.submit(calcf, 'p4')
41     #   exe.shutdown()
42
43     #   concurrent.futuresモジュールによるマルチスレッドの実験
44     #   print('--- By concurrent.futures : [ThreadPoolExecutor] ---')
45     #   exe = futures.ThreadPoolExecutor(max_workers=4)
46     #   f1 = exe.submit(calcf, 'p1')
47     #   f2 = exe.submit(calcf, 'p2')
48     #   f3 = exe.submit(calcf, 'p3')
49     #   f4 = exe.submit(calcf, 'p4')
50     #   exe.shutdown()
51
```

¹⁹⁸詳しくは「6 外部プログラムとの連携」(p.380～)の章にある、円の微分方程式を差分化する方法を参照のこと。

このプログラムの 23 行目以降のコメントを下記のように外して、3 つの実行形態を比較することができる。

- 1) threading モジュールによる実行 → 23～32 行目先頭の # を外す
- 2) concurrent.futures モジュールの ProcessPoolExecutor クラスによる実行 → 35～41 行目先頭の # を外す
- 3) concurrent.futures モジュールの ThreadPoolExecutor クラスによる実行 → 44～50 行目先頭の # を外す

実行した例を次に示す。

1) の実行例（マルチスレッド）

```
--- By threading ---
p1 - time: 7.784013509750366
p3 - time: 7.783469200134277
p2 - time: 8.176703214645386
```

2) の実行例（マルチプロセス）

```
--- By concurrent.futures : [ProcessPoolExecutor] ---
p1 - time: 3.423860549926758
p3 - time: 3.4440042972564697
p4 - time: 3.567429304122925
p2 - time: 3.648512601852417
```

3) の実行例（マルチスレッド）

```
--- By concurrent.futures : [ThreadPoolExecutor] ---
p3 - time: 10.68565034866333
p1 - time: 10.819898128509521
p4 - time: 11.471580266952515
p2 - time: 11.6315176486969
```

※ 実行環境：Python 3.11.2, Intel Corei7-1195G7 2.9GHz (4 コア), 32GB RAM, Windows11 Pro

1) と 3) の形態では、実行する関数 `caclf` の個数が増えると実行時間が伸び、2) の形態では、CPU やコアの数が多くなると実行時間が小さくなる。このことを、関数 `caclf` を実行する個数を変えて確認されたい。

4.5 非同期処理：asyncio

asyncio ライブラリは効率の良い並行処理を実現するものである。asyncio は多くの機能を提供するが、本書ではその内の高レベル API に関する基本的な事柄について解説する。

asyncio による並行処理の考え方は、先の「4.4 マルチスレッドプログラミング」(p.255) で解説したものとは異なり、**コルーチン** (async キーワードで宣言された関数) や **タスク** (Task オブジェクト) といった実行の単位 (Awaitable オブジェクト) を独特のイベントループで管理するものである。具体的には、複数の Awaitable オブジェクトをイベントループに登録し、それらを順番に実行する。そして登録された全ての Awaitable オブジェクトが終了するまでイベントループを実行する。各タスクの切り替えは await の式で制御する。

厳密には asyncio におけるタスクの実行は並行実行ではなく、async 宣言された Awaitable オブジェクトを、1つのスレッド上で await 式によって実行し、それらの一時停止と切替えを管理するものである。特に、ディスクに対する入出力処理や通信といった待ち時間が多く発生する処理を管理する場合にこの実行形態が有利に働くことがある。

4.5.1 コルーチン

コルーチンとは asyncio の並行処理におけるプログラムの単位であり、async キーワードで宣言された関数として実装することができる。(asyncio ライブラリを読み込んでいない状態でも async 宣言は可能である)

例. コルーチンの定義の例

```
>>> async def msg(s): Enter ←コルーチン定義の開始
...     print(s) Enter
...     return '戻り値' Enter
... Enter ←コルーチン定義の終了
```

このようにして定義された msg は通常の変数とは異なり、呼び出すと**コルーチンオブジェクト**を返す。(次の例参照)

例. コルーチンオブジェクトの生成 (先の例の続き)

```
>>> c = msg('コルーチン') Enter ←コルーチンを呼び出す
>>> print(c) Enter ←内容確認
<coroutine object msg at 0x000002423163F1D0> ←コルーチンオブジェクト
```

この例の実行においては msg に定義されたプログラムは動作しておらず、得られたコルーチンオブジェクトをイベントループ上で実行するには次の例のように asyncio の run 関数を使用する。

例. コルーチンの実行 (先の例の続き)

```
>>> import asyncio Enter ←ライブラリの読み込み
>>> r = asyncio.run(c) Enter ←イベントループを起動してコルーチンを実行
コルーチン ←コルーチンからの出力
>>> print(r) Enter ←戻り値の確認
戻り値 ←戻り値が得られている
```

ただし、このような方法では通常の変数を実行する場合と結果的に変わらない。これに関しては次のサンプルプログラム asyncio01.py の実行によって確認できる。

プログラム：asyncio01.py

```
1 # coding: utf-8
2 import asyncio          # ライブラリの読み込み
3
4 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
5     for i in range(3):
6         print(s)
7     return s             # 受け取った引数を返す
8
9 r1 = asyncio.run(msg('コルーチン1')) # この順番でイベントループを
10 r2 = asyncio.run(msg('コルーチン2')) # 起動してコルーチンを
11 r3 = asyncio.run(msg('コルーチン3')) # 実行する
12
13 print('---\n',r1,r2,r3)
```

このプログラムを実行すると次のような出力となる。

出力)

```
コルーチン 1
コルーチン 1
コルーチン 1          ← 1  つ目のコルーチンが終了した後で
コルーチン 2          ← 2  つ目のコルーチンが起動する
コルーチン 2
コルーチン 2          ← 2  つ目のコルーチンが終了した後で、
コルーチン 3          ← 3  つ目のコルーチンが起動する
コルーチン 3
コルーチン 3          ← 3  つ目のコルーチンが終了
—
コルーチン 1 コルーチン 2 コルーチン 3    ←それぞれの戻り値を出力
```

この実行例からわかるように、3つのコルーチンが順番に実行されており、並行処理とはなっていないことが確認できる。このような記述では `run` によって3つのイベントループが順番に起動、終了され、それぞれのイベントループ内でコルーチンが実行されるので並行処理のような形にはならない。

4.5.2 コルーチンとタスクの違い

1つのイベントループ上で複数の `Awaitable` オブジェクトの実行を管理するには、それらを**タスク**の形にしなければならない。すなわち、コルーチン自体にはイベントループ上での実行に関する管理の機能がなく、イベントループ上でのスケジューリングの対象とするには、そのための機能を持った**タスク**にする必要がある。タスクを作成するには `create_task` 関数を使用できる。

書き方： `create_task(コルーチン)`

「コルーチン」からタスクを作成して返す。

4.5.2.1 複数のタスクの並行実行

複数のタスクを1つのイベントループで並行実行するには、それら複数のタスクを **await 式**で起動するためのコルーチンを1つ実装して、それを `run` によって起動するとよい。

次に、`await` 式によるタスクの実行開始、一時停止、切り替えについて説明する。

4.5.3 `await` によるタスクの一時停止と切替え

`asyncio` の `sleep` 関数を使用すると、実行中の `Awaitable` オブジェクトの実行を一時停止して、実行制御をイベントループ上の実行可能な次の `Awaitable` オブジェクトに移すことができる。このことをサンプルプログラム `asyncio02.py` の実行を通して解説する。

プログラム： `asyncio02.py`

```
1  # coding: utf-8
2  import asyncio          # ライブラリの読み込み
3
4  async def msg(s):        # コルーチン：引数に与えた値を3回出力する
5      for i in range(3):
6          print(s)
7          await asyncio.sleep(1) # ←このコルーチンを一時停止して
8          return s          # 次のタスクに制御を移す
9
10 async def exe():         # タスクを投入して実行するコルーチン
11     t1 = asyncio.create_task( msg('コルーチン1') ) # 3つのコルーチンを
12     t2 = asyncio.create_task( msg('コルーチン2') ) # それぞれタスクに
13     t3 = asyncio.create_task( msg('コルーチン3') ) # している。
14     await t1; await t2; await t3                  # 順番に登録して実行
15
16 asyncio.run( exe() )    # イベントループの起動
```

解説)

4～8行目でコルーチン `msg` が定義されている。このコルーチンは受け取った引数を3回出力するものであるが、毎回の出力の直後に `asyncio.sleep(1)` を `await` 式によって実行している。これにより、当該コルーチンは実行が一時

停止され、1 秒間に渡って休止状態となる。同時に、イベントループの上にある次の実行可能な Awaitable オブジェクトに実行の制御が移る。

10～14 行目に定義されているコルーチン `exe` は `msg` を 3 つのタスクにして (11～13 行目) イベントループに登録して (14 行目) 実行を開始するものである。この際、`create_task` 関数によってコルーチン `msg` をタスクオブジェクト (`asyncio.Task` クラス) にしている。タスクオブジェクトは生成直後は実行されず、14 行目にある `await` 式によってイベントループに登録されて実行される。

このプログラムの実行結果の出力を次に示す。

出力)

```
コルーチン 1
コルーチン 2
コルーチン 3      ←ここで 1 秒弱停止
コルーチン 1
コルーチン 2
コルーチン 3      ←ここで 1 秒弱停止
コルーチン 1
コルーチン 2
コルーチン 3      ←ここで 1 秒弱停止した後、プログラムが終了
```

この実行例からわかるように、3 つのタスクが同時に実行されているように見える。

注意) 次の `asyncio02-2.py` のように実装すると並行実行のようにはならないことに注意すること。

プログラム: `asyncio02-2.py` (良くない実装)

```
1  # coding: utf-8
2  import asyncio          # ライブラリの読み込み
3
4  async def msg(s):        # コルーチン: 引数に与えた値を3回出力する
5      for i in range(3):
6          print(s)
7          await asyncio.sleep(1) # ←このコルーチンを一時停止して
8      return s             # 次のタスクに制御を移す
9
10 async def exe():         # タスクを投入して実行するコルーチン
11     t1 = asyncio.create_task(msg('コルーチン1')); await t1    # タスクの生成と
12     t2 = asyncio.create_task(msg('コルーチン2')); await t2    # それらの実行を
13     t3 = asyncio.create_task(msg('コルーチン3')); await t3    # この方法で試みる
14
15 asyncio.run(exe())       # イベントループの起動
```

このプログラムの実行結果の出力を次に示す。

出力)

```
コルーチン 1      ←ここで約 1 秒停止
コルーチン 1      ←ここで約 1 秒停止
コルーチン 1      ←ここで約 1 秒停止して 1 つ目のコルーチンが終了
コルーチン 2      ←ここで約 1 秒停止
コルーチン 2      ←ここで約 1 秒停止
コルーチン 2      ←ここで約 1 秒停止して 2 つ目のコルーチンが終了
コルーチン 3      ←ここで約 1 秒停止
コルーチン 3      ←ここで約 1 秒停止
コルーチン 3      ←ここで約 1 秒停止した後、プログラムが終了
```

各タスクが並行実行のようにはならず、順番に実行されていることがわかる。複数の非同期処理を並行処理のように実行するには、`exe` の末尾にまとめて `await` 式で実行する必要がある。

4.5.4 タスク登録の簡便な方法

`asyncio` の `gather` 関数を使用すると、タスクの生成と登録を更に簡単な形で実現できる。

書き方: `gather(Awaitable オブジェクト 1, Awaitable オブジェクト 2, ...)`

引数に与えた Awaitable オブジェクト (コルーチン、タスクなど) をその順序でタスクとしてイベントループに登録して実行する。

`gather` 関数を用いて先のプログラム `asyncio02.py` と同様の処理を実現するプログラム `asyncio03.py` を示す。

プログラム：asyncio03.py

```
1 # coding: utf-8
2 import asyncio          # ライブラリの読み込み
3
4 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
5     for i in range(3):
6         print(s)
7         await asyncio.sleep(1) # ←このコルーチンを一時停止して
8         return s            # 次のタスクに制御を移す
9
10 async def exe():        # タスクを投入して実行するコルーチン
11     g = await asyncio.gather( # 実際にタスクを実行する
12         msg('コルーチン1'),
13         msg('コルーチン2'),
14         msg('コルーチン3')
15     )
16     print(g)            # 戻り値の確認
17
18 asyncio.run( exe() )    # イベントループの起動
```

gather 関数の戻り値がリストとして変数 g に得られ（11～15 行目）各コルーチンからの戻り値が要素となる。

4.5.5 イベントループと run 関数

イベントループは1つのスレッド内に1つだけ実行できるものである。従って、一度 run 関数で非同期処理が開始された後は、そのイベントループが終了するまで次の run 関数は実行できない。先に示したサンプルプログラム asyncio02.py～asyncio03.py において、exe を最上位のコルーチンとして run 関数で実行し、exe の内部で複数のタスクを登録して実行するという形式をとっているのはそのような事情による。

4.5.5.1 スレッド毎に実行されるイベントループ

イベントループは通常、run 関数の実行によって自動的に作成されるが、asyncio の new_event_loop 関数によって明示的に作成（ProactorEventLoop オブジェクトを作成）することもできる。また、イベントループは異なるスレッド毎に作成することもできるので、スレッド毎に異なるイベントループを同時に実行することもできる。このことを次のサンプルプログラム asyncio04.py で示す。

プログラム：asyncio04.py

```
1 # coding: utf-8
2 import asyncio
3 import threading
4
5 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
6     for i in range(2):
7         print(s)
8         await asyncio.sleep(1) # ←このコルーチンを一時停止して
9         return s            # 次のタスクに制御を移す
10
11 def startLoop(eloop,rn): # スレッド内でイベントループを開始する
12     eloop.run_until_complete( msg(rn) )
13
14 eloopA = asyncio.new_event_loop() # イベントループ(1)
15 eloopB = asyncio.new_event_loop() # イベントループ(2)
16
17 # スレッドを2つ生成して各々でイベント管理する
18 tA = threading.Thread( target=startLoop, args=(eloopA,'コルーチン：スレッドA') )
19 tB = threading.Thread( target=startLoop, args=(eloopB,'コルーチン：スレッドB') )
20 # スレッドの開始と終了待ち
21 tA.start();          tB.start()
22 tA.join();            tB.join()
```

このプログラムの 14,15 行目で new_event_loop 関数によって別々のイベントループを作成している。またそれらを 18,19 行目でスレッド tA, tB に与えている。スレッドとして起動する関数 startLoop はイベントループと文字列を受け取り、run_until_complete メソッドによってイベントループを起動する。

書き方： イベントループ.run_until_complete(コルーチン)

「コルーチン」に非同期処理の最上位のコルーチンを与える。このメソッドは非同期処理が終了した際にコルーチンの戻り値を返す。run_until_complete は、asyncio.run と同様の処理を、指定したイベントループに対して実行する際に用いる。

このプログラムをスクリプトとして実行した様子を次に示す。

例. asyncio04.py の実行

コルーチン：スレッド A
コルーチン：スレッド B
コルーチン：スレッド A
コルーチン：スレッド B

2つのイベントループが平行して動作していることがわかる。

勿論であるが、各々のスレッド内で run 関数によってイベントループを自動的に作成することもできる。次の asyncio04-2.py は更に簡潔な記述で先のプログラムと同等の処理を行うものである。

プログラム：asyncio04-2.py

```
1  # coding: utf-8
2  import asyncio
3  import threading
4
5  async def msg(s):          # コルーチン：引数に与えた値を3回出力する
6      for i in range(2):
7          print(s)
8          await asyncio.sleep(1) # ←このコルーチンを一時停止して
9      return s               # 次のタスクに制御を移す
10
11 def startLoop(rn): # スレッド内でイベントループを生成して開始する
12     asyncio.run( msg(rn) )
13
14 # スレッドを2つ生成して各々でイベント管理する
15 tA = threading.Thread( target=startLoop, args=('コルーチン：スレッドA',) )
16 tB = threading.Thread( target=startLoop, args=('コルーチン：スレッドB',) )
17 # スレッドの開始と終了待ち
18 tA.start();          tB.start()
19 tA.join();           tB.join()
```

4.5.5.2 実行中のイベントループを調べる方法

get_running_loop 関数を用いると、現行のスレッドで実行中のイベントループを調べることができる。

書き方： asyncio.get_running_loop()

この関数は、イベントループオブジェクト（ProactorEventLoop オブジェクト）を返す。実行中のイベントループが存在しない場合は例外（RuntimeError）が発生する。

参考)

get_running_loop 関数と類似のものとして get_event_loop も存在するが、これは将来の Python の版において廃止されることが予定されている。

4.5.6 理解を深めるためのサンプル

時間かかる処理を非同期処理（async, await）で管理する方法について考える。

例. 時間のかかる処理

```
>>> import random  [Enter]
>>> while True:    [Enter]
...     r = random.random() [Enter]    ←乱数を生成し
...     if r < 2**(-26): [Enter]        それが 2-26 未満であれば
...     break      [Enter]            終了するループ
... [Enter]        ←反復処理の記述の終了
```

このプログラムは非常に小さな乱数（発生確率の非常に低い乱数）が得られるまで乱数生成を繰り返すもので、終了まで時間がかかる例である。（実際に試されたい）このような処理を複数起動して並行実行するサンプルを次の asyncio07.py

に示す。

プログラム：asyncio07.py

```
1 # coding: utf-8
2 import asyncio
3 import random
4 import time
5
6 # v未満の乱数が出るまで試行を続けるコルーチン
7 async def long_task( n, v ):
8     while True:
9         r = random.random()
10        if r < v:
11            break
12        await asyncio.sleep(0)
13    print( n, r )
14
15 # 上記処理を繰り返すコルーチン
16 async def asyncIteration( n, v, i ):
17     for x in range(i):
18         await long_task( n, v ) # 各回の処理の完了を待つ
19
20 # イベントループに渡す最上位のコルーチン
21 async def exe():
22     await asyncio.gather(
23         asyncIteration('task1',2**(-21),2), # 2つの処理が
24         asyncIteration('task2',2**(-20),4)  # 並行実行される形
25     )
26
27 # 非同期処理の実行
28 print('開始')
29 t1 = time.time()
30 asyncio.run( exe() )
31 t2 = time.time()
32 print('終了:',t2-t1,'秒')
```

このプログラムは先の例と同じ処理を並行実行するものである。関数 `long_task(n,v)` は非同期処理のためのコルーチンとして定義されており、タスクの識別名 `n` と乱数の終了条件の値 `v` を引数に取る。このコルーチンでは、多数回反復される `while` の処理の中の

```
await asyncio.sleep(0)
```

において実行の管理をイベントループに移している。待ち時間を0にしているが、これは実行待ちをさせるのが目的ではなく、`while` による反復で毎回イベントループに制御を渡すのが目的である。すなわち、この行の記述がないと `while` ループが終了するまで当該スレッドが他のタスクの実行をブロックしてしまう。

コルーチン `asyncIteration` は時間のかかる `long_task` を指定した回数（引数 `i`）繰り返すものである。また、コルーチン `exe` はイベントループで実行する2つのタスクを起動するものである。

このプログラムを実行した際の出力の例を次に示す。

```
開始
task2 2.723247869429457e-07
task2 6.862664404527763e-07
task2 9.22241627954179e-07
task1 3.4535740955909944e-07
task2 5.80478085154823e-07
task1 2.7896666210391885e-07
終了: 7.798398494720459 秒
```

2つのタスクが互いにブロックせずに実行されている様子がわかる。

4.5.6.1 非同期処理とイベントループの概観

複数のタスクの実行がイベントループ上で管理される様子を図35に示す。

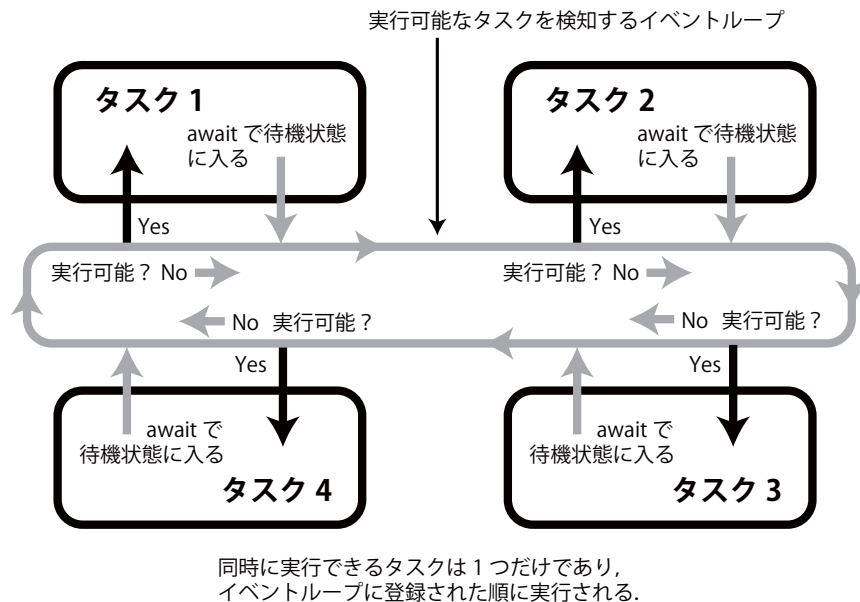


図 35: イベントループで監視されるタスクの概観

4.5.7 スレッドを非同期処理で管理する方法

イベントループに対する `run_in_executor` メソッドを使用すると、Awaitable でない通常の関数をメインのスレッドとは別のスレッドとして起動して非同期処理の枠組みで管理することができる。

書き方： イベントループ.`run_in_executor`(実行器, 関数, 引数並び…)

「関数」に「引数並び…」を与えて実行するための `asyncio.Future` オブジェクトを返す。この関数を実行する際に使用される「実行器」(`concurrent.futures.ThreadPoolExecutor` など)を与える。「実行器」として `None` を与えるとシステムが用意したものが採用される。

このメソッドで得られた `Future` オブジェクトをコルーチンにしたものをタスクに変換すると、イベントループ上の非同期処理として実行することができる。このことを次のサンプルプログラム `asyncio05.py` で解説する。

プログラム： `asyncio05.py`

```

1  # coding: utf-8
2  import asyncio, time
3  from concurrent import futures
4
5  def normalf(s):          # 通常の関数
6      for i in range(3):
7          print(s)
8          time.sleep(1)
9      return s
10
11 # Futureオブジェクトを実行するコルーチンの作成
12 async def fut2cor(f):
13     c = await f
14     return c
15
16 # スレッド実行器
17 e = futures.ThreadPoolExecutor(max_workers=2)
18
19 async def exe():          # タスクを投入して実行するコルーチン
20     eLoop = asyncio.get_running_loop() # イベントループを取得
21     # 通常の関数のFutureオブジェクトを作成
22     f1 = eLoop.run_in_executor(e, normalf, '通常の関数1')
23     f2 = eLoop.run_in_executor(e, normalf, '通常の関数2')
24     # Futureオブジェクトをタスクに変換
25     t1 = asyncio.create_task(fut2cor(f1))
26     t2 = asyncio.create_task(fut2cor(f2))
27     # 両方のタスクを同時に実行
28     r = await asyncio.gather(t1, t2)
29     # 戻り値の確認
30     print(r)

```

```

31     # 実行器の終了
32     e.shutdown()
33
34 asyncio.run(exe()) # イベントループの起動

```

解説)

このプログラムの関数 `normalf` は、引数に与えられた値を出力して1秒間待つ処理を3回繰り返すもので、これをイベントループ上で実行する。

`run_in_executor` はイベントループに対して実行するものなので、システムが用意したイベントループを20行目で取得している。22,23行目で関数 `normalf` を実行するためのFutureオブジェクト `f1`, `f2` を作成している。そして、それを関数 `fut2cor` でコルーチンに変換し、`create_task` 関数でタスク `t1`, `t2` にしている。(25,26行目)

このプログラムの実行結果を次に示す。

例. `asyncio05.py` をスクリプトとして実行

```

通常関数 1
通常関数 2
通常関数 1
通常関数 2
通常関数 1
通常関数 2
['通常関数 1', '通常関数 2']

```

スレッド上で実行される関数自体は非同期処理を意識せずに実装できるので、それを非同期処理の枠組みの上で管理したい場合にこの方法が有効である。

勿論のことではあるが、コルーチンをタスクにしたものと、スレッドをタスクにしたものを同時にイベントループ上で実行することができる。そのことを次の `asyncio06.py` で示す。

プログラム: `asyncio06.py`

```

1  # coding: utf-8
2  import asyncio, time
3  from concurrent import futures
4
5  def normalf(s):          # 通常関数
6      for i in range(3):
7          print(s)
8          time.sleep(1)
9      return s
10
11 async def coroutine(s):   # コルーチン
12     for i in range(3):
13         print(s)
14         await asyncio.sleep(1)
15     return s
16
17 # Futureオブジェクトを実行するコルーチンの作成
18 async def fut2cor(f):
19     c = await f
20     return c
21
22 # スレッド実行器
23 e = futures.ThreadPoolExecutor(max_workers=1)
24
25 async def exe():          # タスクを投入して実行するコルーチン
26     eLoop = asyncio.get_running_loop() # イベントループを取得
27     # 通常関数のFutureオブジェクトを作成
28     f = eLoop.run_in_executor(e, normalf, '通常関数')
29     # タスクを作成
30     t1 = asyncio.create_task(fut2cor(f))          # スレッドのタスク
31     t2 = asyncio.create_task(coroutine('コルーチン')) # コルーチンのタスク
32     # 両方のタスクを同時に実行
33     r = await asyncio.gather(t1, t2)
34     # 戻り値の確認
35     print(r)
36     # 実行器の終了
37     e.shutdown()
38

```

このプログラムの実行結果を次に示す。

例. `asyncio06.py` をスクリプトとして実行

```
通常関数
コルーチン
通常関数
コルーチン
通常関数
コルーチン
['通常関数', 'コルーチン']
```

`normalf` と `coroutine` がタスクとして同時に実行されている様子がわかる。

4.5.8 非同期のコンソール入力を実現するためのライブラリ：aioconsole

有志のエンジニア Vincent Michel 氏が開発して公開しているライブラリ `aioconsole`¹⁹⁹ を用いると、`input` 関数と同様のコンソール入力を非同期に実行することができる。このライブラリは Python 言語処理系の標準ライブラリではなく、`pip` コマンドなどでシステムに導入する必要がある。

非同期のコンソール入力には `ainput` 関数を用いる。

書き方： `ainput(プロンプト)`

「プロンプト」に与えた文字列をコンソールに表示して入力を促し、コンソールから入力された文字列を返す。

`ainput` 関数を使用したサンプル `asyncio08.py` を次に示す。

プログラム： `asyncio08.py`

```
1  # coding: utf-8
2  import asyncio
3  import aioconsole
4
5  # 非同期のタスク1
6  async def msg(s):
7      for i in range(3):
8          print(s)
9          await asyncio.sleep(4)
10     return s
11
12 # 非同期のタスク2
13 async def asyncInput():
14     while True:
15         r = await aioconsole.ainput('endで終了> ')
16         print('入力値:', r)
17         await asyncio.sleep(4)
18         if r == 'end': break
19
20 # タスク投入用コルーチン
21 async def exe():
22     g = await asyncio.gather(
23         msg('文字列表示'),      # タスク1
24         asyncInput()             # タスク2
25     )
26
27 asyncio.run(exe()) # イベントループで実行
```

このサンプルでは2つのコルーチン `msg` と `asyncInput` が定義され、それらが非同期のタスクとして実行される。`msg` は引数に与えられた文字列を3回出力する。(毎回の出力の後で4秒待機する) `asyncInput` は非同期に実行され、他のタスクをブロックすることなくコンソールからの入力を受け付ける。このサンプルの実行例を次に示す。

¹⁹⁹<https://github.com/vxgmichel/aioconsole>

例. asyncio08.py の実行

```
文字列表示          ← msg による出力
end で終了> abcd  Enter ←キーボードからの入力
入力値:  abcd
文字列表示          ← msg による出力
end で終了> end   Enter ←キーボードからの入力
入力値:  end
文字列表示          ← msg による出力
```

msg と asyncInput が互いに相手をブロックせずに動作している様子がわかる。

4.5.9 永続するイベントループ上でのタスク管理

ここまでの解説では、イベントループ上のタスクが終了した時点で当該イベントループも終了する形の実装を示してきた。この形とは別に、永続するイベントループに対して動的にタスクを投入するという形式のプログラミング²⁰⁰も実際に多く行われている。ここでは、永続するイベントループとその上でのタスクの管理の方法について解説する。

4.5.9.1 永続するイベントループ

イベントループに対して run_forever メソッドを実行するとそのイベントループの動作が永続的になる。

書き方： イベントループ.run_forever()

また、永続するイベントループを停止するには stop メソッドを使用する。

書き方： イベントループ.stop()

動作を終了して停止したイベントループは close メソッドで閉じておくこと。

書き方： イベントループ.close()

永続するイベントループに対して動的にタスクを投入する例を asyncio09.py で示す。

プログラム：asyncio09.py

```
1  # coding: utf-8
2  import asyncio
3  import aioconsole
4
5  #--- タスクの定義 -----
6  # 非同期のタスク：メッセージを3回出力
7  async def msg( n, s ):
8      for i in range(3):
9          print( '\t'*n + s )
10         await asyncio.sleep(1)
11     return s
12
13 # 非同期のタスク：コンソール入力
14 async def asyncInput():
15     while True:
16         r = await aioconsole.ainput('endで終了> ')
17         if r == 'end':
18             asyncio.get_running_loop().stop()    # イベントループの終了
19             print('終了します。')
20             break
21         else:
22             print(' 入力値:',r)
23
24 # 最初に起動される非同期タスク
25 async def theFirst():
26     asyncio.create_task( msg(0,'タスク1') )      # 動的にタスクを投入(1)
27     await asyncio.sleep( 0.5 )
28     asyncio.create_task( msg(1,'タスク2') )      # 動的にタスクを投入(2)
29     await asyncio.sleep( 3 )
30     asyncio.create_task( asyncInput() )          # 動的にタスクを投入(3)
31
32 #--- イベントループの作成，最初のタスクの登録，永続実行 -----
```

²⁰⁰JavaScript 言語処理系（Web ブラウザのエンジンや Node.js など）におけるプログラミングもこのスタイルである。

```

33 | eloop = asyncio.new_event_loop()      # イベントループを作成して
34 | asyncio.set_event_loop( eloop )      # それを現行のスレッドにアタッチする。
35 |
36 | eloop.create_task( theFirst() )       # 最初のタスクをイベントループに登録して
37 | eloop.run_forever()                  # そのイベントループを永続実行する。
38 |
39 | eloop.close()                        # 終了後はイベントループを閉じる。

```

このサンプルでは、メッセージの出力と1秒間の待機を3回繰り返すコルーチン `msg` と、コンソールからの非同期入力を受け付けるコルーチン `asyncInput` が定義されている。これらが、別のコルーチン `theFirst` から順次起動される形になっている。`asyncInput` の入力において 'end' と入力されるとイベントループが `stop` メソッドにより停止され、プログラム全体が終了する。

`asyncio09.py` を実行した際のコンソールの表示を次に示す。

例. ターミナルウィンドウで `asyncio09.py` を実行した様子

```

タスク 1
  タスク 2                ← 2つのタスク msg が互いに
タスク 1                  ブロックせずに実行されている
タスク 1                  タスク 2
  タスク 2                ←ここで2つのタスク msg が終了した
end で終了> end   Enter    ← 終了を指示
終了します。

```

参考)

このサンプルでは、イベントループ `eloop` を作成した後で、

```
asyncio.set_event_loop( eloop )
```

として現行のスレッドにそれをアタッチしているが、当サンプルではこの行を省略することができる。

4.5.9.2 タスクの一覧取得とタスクのキャンセル

`all_tasks` 関数を用いると、イベントループの上で実行されているタスクの一覧 (Set オブジェクト) を取得することができる。

書き方: `asyncio.all_tasks(イベントループ)`

「イベントループ」 (ProactorEventLoop オブジェクト) の上で実行されているタスク (Task オブジェクト) を要素とする Set オブジェクトを返す。

実行中のタスクを終了する (キャンセルする) には `cancel` メソッドを使用する。

書き方: `タスク.cancel()`

「タスク」 (Task オブジェクト) を強制終了する。この際に警告が発生することがあるのでそれを適切にハンドリングすること。

次に示すサンプル `asyncio010.py` は、コンソールからの非同期の入力を受けて新規のタスクをイベントループに随時投入するものである。

プログラム: `asyncio10.py`

```

1 | # coding: utf-8
2 | import asyncio
3 | import aioconsole
4 |
5 | # --- タスクの定義 -----
6 | # 非同期のタスク: 無制限の繰り返し
7 | async def tsk():
8 |     while True: await asyncio.sleep(1)
9 |
10 | # 非同期のタスク: コンソール入力
11 | async def asyncInput():
12 |     el = asyncio.get_running_loop()
13 |     while True:
14 |         r = await aioconsole.ainput('endで終了> ')
15 |         if r == 'end':

```



```

16         for t in asyncio.all_tasks( el ):
17             t.cancel()                                # タスクを強制的にキャンセル
18             try:                                       # その際の例外処理
19                 await t
20             except asyncio.CancelledError:
21                 pass                                  # 必要に応じてここで例外処理を実行する
22             asyncio.get_running_loop().stop()         # イベントループの終了
23         print('終了します. ')
24         break
25     elif r == 'show':
26         print( '実行中>\t', end='' )
27         for t in asyncio.all_tasks( el ):
28             print( t.get_name(), sep='', end='\t' ) # 実行中タスクの一覧出呂っく
29         print()
30     elif r == 'new':
31         asyncio.create_task( tsk() )                 # 動的にタスクを投入
32     else:
33         print(' 入力値:',r)
34
35 # 最初に起動される非同期タスク
36 async def theFirst():
37     asyncio.create_task( asyncInput() )
38
39 #--- イベントループの作成, 最初のタスクの登録, 永続実行 -----
40 eloop = asyncio.new_event_loop()                    # イベントループを作成して
41
42 eloop.create_task( theFirst() )                      # 最初のタスクをイベントループに登録して
43 eloop.run_forever()                                  # そのイベントループを永続実行する.
44
45 eloop.close()                                        # 終了後はイベントループを閉じる.

```

このサンプルでは、コンソールから 'show' と入力すると実行中のタスクの一覧（タスク名の一覧）を表示する。この際に all_tasks メソッドが使用される。また、タスク名の取得には当該タスクに対して get_name を実行する。

コンソールから 'new' と入力するとコルーチン tsk をタスクとしてイベントループに追加し、'end' と入力すると実行中のタスクを全て強制終了してプログラム全体を終了する。

実行中のタスクを cancel メソッドで強制的に終了するとしばしば警告（asyncio.CancelledError）が発生するので、これを適切にハンドリングすること。

asyncio10.py を実行した際のコンソールの表示を次に示す。

例. ターミナルウィンドウで asyncio10.py を実行した様子

```

end で終了> show  [Enter]    ←状態確認
実行中> Task-2      ←この段階では asyncInput だけが実行されている

end で終了> new  [Enter]    ←新規タスク投入
end で終了> show  [Enter]    ←状態確認
実行中> Task-5 Task-2      ←タスクが 1 つ増えている

end で終了> new  [Enter]    ←新規タスク投入
end で終了> show  [Enter]    ←状態確認
実行中> Task-5 Task-8 Task-2 ←タスクが 1 つ増えている

end で終了> end  [Enter]    ←終了を指示
終了します.

```

■ タスクの名前について

イベントループに投入されたタスクには自動的に名前が付与される。先の実行例で得られた

Task-5 Task-8 Task-2

のような出力には各タスクに自動的に命名されたものが見られる。タスク投入時に意図して名前を与える場合は、キーワード引数 'name=' にそれを指定して、

```
asyncio.create_task( コルーチン, name=タスク名 )
```

のように記述する。

4.5.10 非同期のイテレーション

for 文による反復処理に `async` キーワードを付けることで**非同期のイテレーション**を実現することができる。

書き方： `async for 変数 in 非同期のイテラブル:`
(反復対象の処理)

「非同期のイテラブル」(後述) から要素を順次「変数」に受け取って「反復対象の処理」を実行する。「変数」に値を受け取る際に `await` による待機が起こる場合、イベントループに実行の制御が移される。

4.5.10.1 非同期のイテラブル

要素の取得(あるいは生成)に時間がかかる(`await` による待機が起こる)イテラブルは**非同期のイテラブル**²⁰¹ として扱うことができる。非同期のイテラブルには様々なものがあり、ファイル入出力や通信における入力ストリームを非同期の形で扱うイテラブルが代表的な例である。また、次に説明する**非同期のジェネレータ**、**非同期のイテレータ**もこの範疇に入る。

■ 非同期のジェネレータ

非同期のジェネレータは `async` キーワードで宣言された**ジェネレータ**²⁰² である。非同期のジェネレータの例を次のサンプル `asyncio11.py` で示す。

プログラム： `asyncio11.py`

```
1 # coding: utf-8
2 import asyncio
3
4 # コルーチン1
5 async def msg():
6     for x in range(4):
7         await asyncio.sleep(1)
8         print( 'タスク1' )
9
10 # コルーチン2：非同期のジェネレータ
11 async def aGen():
12     for x in ['a','b']:
13         await asyncio.sleep(2)
14         yield x
15
16 # タスクを投入するコルーチン
17 async def exe():
18     asyncio.create_task( msg() )      # タスク1
19     async for x in aGen():             # 非同期の反復処理
20         print( '\t非同期に値を取得:',x )
21
22 asyncio.run( exe() )                 # イベントループで実行
```

このサンプルでは非同期のジェネレータ `aGen` がコルーチンの形で定義されている。このジェネレータは2つの要素 `'a'`、`'b'` を順次返すものであり、値を返す前に2秒の待ち時間がある。また、これとは別にコルーチン `msg` が定義されており、1秒の待ち時間の後でメッセージを出力することを4回繰り返す。

`aGen` は非同期のイテレーション(`async for` 文)で使用され、要素取得までの待ち時間の間も他のタスク `msg` をブロックしない。

このサンプルを実行した際の出力の例を次に示す。

例. ターミナルウィンドウで `asyncio11.py` を実行した様子

タスク1	← <code>msg</code> による出力
非同期に値を取得: a	← <code>async for</code> 文内の <code>print</code> による出力
タスク1	← <code>msg</code> による出力
タスク1	← <code>msg</code> による出力
非同期に値を取得: b	← <code>async for</code> 文内の <code>print</code> による出力
タスク1	← <code>msg</code> による出力

²⁰¹非同期のコンテナと呼ぶこともある。

²⁰²後の「4.7 ジェネレータ」(p.297)で詳しく解説する。

msg のタスクと非同期のイテレーションの処理が互いにブロックされずに実行されている様子がわかる。

■ 非同期のイテレータ

非同期のイテレータとは、要素を返す処理を非同期に行うイテレータである。基本的な実装方法は先の「2.9.7 イテレータのクラスを実装する方法」(p.169) に似ており、2つのメソッド `__aiter__`、`__anext__` の実装による。特に `__anext__` メソッドは値を返す処理を非同期に実行するので `async` キーワードを付けて宣言する。

非同期のイテレータの例を次のサンプル `asyncio12.py` で示す。これは先の `asyncio11.py` と同等の処理を実行する（先の実行例と同様の出力を得る）ものである。

プログラム： `asyncio12.py`

```
1  # coding: utf-8
2  import asyncio
3
4  # コルーチン1
5  async def msg():
6      for x in range(4):
7          await asyncio.sleep(1)
8          print( 'タスク1' )
9
10 # 非同期のイテレータのクラス
11 class AsyncIter:
12     def __init__(self):                # コンストラクタ
13         self.Q = ['a','b']
14         self.cur = 0
15     def __aiter__(self):
16         return self
17     async def __anext__(self):          # 次の要素を返すメソッド（非同期処理）
18         if self.cur > 1:                # イテレーションが終了している場合に
19             raise StopAsyncIteration   # この例外を起こす
20         else:
21             await asyncio.sleep(2)
22             r = self.Q[self.cur]
23             self.cur += 1
24             return r
25
26 # タスクを投入するコルーチン
27 async def exe():
28     asyncio.create_task( msg() )        # タスク 1
29     A = AsyncIter()                    # 非同期のイテレータの作成
30     async for x in A:                  # 非同期の反復処理
31         print( '\t非同期に値を取得:',x )
32
33 asyncio.run( exe() )                  # イベントループで実行
```

このサンプルでは非同期のイテレータのクラス `AsyncIter` が定義されている。コルーチン `exe` の中でこのクラスのインスタンス `A` が作成されて非同期のイテレーションに使用されている。

4.5.11 非同期のファイル入出力： `aiofiles`

`aiofiles` ライブラリは、`asyncio` ライブラリと連携して非同期のファイル操作を行う機能を提供する。Python 処理系の通常のファイル操作は同期処理であり、処理が完了するまで当該スレッドをブロックする。`aiofiles` はこれを回避するために、ファイル操作を別のスレッドプールに委譲する形で非同期のファイル操作を行う。

`aiofiles` ライブラリは Python の標準ライブラリではなく、サードパーティ²⁰³ が提供するものであり、これを使用するには `pip` コマンドなどで Python 言語処理系の環境に予め導入しておく必要がある。

`aiofiles` ライブラリが提供する API は、Python の標準的なファイル操作の API の場合と同様の記述ができるように配慮されている。また、標準ライブラリである `os` モジュールが提供するファイル操作関連の API と同じ名称の API が多数提供されており、`os` モジュールの場合と同様の記述方法で利用できる点も大きな特徴である。

²⁰³Tin Tvrtkovic 氏によって開発された。(<https://pypi.org/project/aiofiles/>)

aiofiles によるファイル入出力は「2.7 入出力」(p.103) で解説した API と類似の用法で実現できる。すなわち、open 関数でファイルを開いてファイルオブジェクトを作成し、それに対して read, write メソッドを実行して入力、出力を行い、最後に close メソッドを実行してファイルを閉じる。(それらの関数やメソッドを await 式として実行する)

本書では使用例を挙げて aiofiles の最も基本的な API について解説する。提供されている各 API とその詳細に関しては aiofiles の公式インターネットサイトなどを参照のこと。

4.5.11.1 サンプルプログラム

次に示す aiofiles01.py では、2つのタスク aFileIOr2(), msg() がイベントループ上で互いにブロックすることなく実行される。aFileIOr2() のタスクは、サイズの大きなファイルを作成して直後にそれを読み込むことを2回繰り返すというもので、入出力に伴う待ち時間が発生するものである。また msg() のタスクは一定の時間間隔でメッセージを繰り返しコンソールに出力するもので、1つ目のタスクが実行を終えた後で終了する。

プログラム：aiofiles01.py

```
1  # coding: utf-8
2  import asyncio
3  import aiofiles
4
5  flg = True      # 共有変数：aFileIOr2 が実行中はTrue, 終わればFalse
6
7  # ファイル出力と読み込みを2回繰り返す処理
8  async def aFileIOr2():
9      global flg
10     for _ in range(2):
11         print('ファイルの作成')
12         # ファイルの作成と出力
13         f = await aiofiles.open('aiofiles01.bin', 'w')
14         for n in range(50000):      # 50,000回出力 (反復回数は適宜調整する)
15             c = chr( n%26 + 97 )    # a~zの文字を作成
16             await f.write(c)        # 1文字ずつ出力
17             if c == 'z': await f.write('\n')
18         await f.close()
19         print('ファイルの読み込み')
20         # 上で作成したファイルの読み込み
21         f = await aiofiles.open('aiofiles01.bin', 'r')
22         while True:
23             c = await f.read(1)
24             if not c: break
25         await f.close()
26     flg = False
27
28 # 定期的にメッセージを出力する処理
29 async def msg():
30     global flg
31     while flg:      # aFileIOr2()が終了するまで繰り返す
32         await asyncio.sleep(2.5)    # 待ち時間は適宜調整する
33         print('\t\t別のタスク')
34
35 # タスク投入用コルーチン
36 async def exe():
37     t1 = asyncio.create_task( aFileIOr2() )
38     t2 = asyncio.create_task( msg() )
39     await t1;    await t2
40
41 asyncio.run( exe() )    # 実行
```

このプログラムの実行例を次に示す。

例. aiofiles01.py を OS のターミナルウィンドウで実行した際の出力

```
ファイルの作成
      別のタスク
ファイルの読み込み
      別のタスク
ファイルの作成
      別のタスク
ファイルの読み込み
      別のタスク
      別のタスク
```

このように、2つのタスク aFileIOr2(), msg() がイベントループ上で互いにブロックすることなく実行されているのがわかる。

このプログラムの実行によって作成されるファイル aiofiles01.bin は次のような内容となる。

```
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
⋮
(以下同様)
⋮
```

上記プログラム中ではファイルを開く処理に open 関数を使っており、引数の仕様も標準の open 関数に準じている。ただし、aiofiles の open 関数が返すファイルオブジェクトのクラスは表 37 に示すように、標準の open 関数が返すものとは異なる。

表 37: open 関数が返すファイルオブジェクトのクラス (一部)		
	テキスト	バイナリ
入力用	AsyncTextIOWrapper	AsyncBufferedReader
出力用	(同上)	AsyncBufferedIOBase

ここで示したサンプル aiofiles01.py は、ファイルの入出力の待ち時間を強調するために、敢えて 1 バイトずつ入出力を行ったが、実用的な局面では長いデータを適宜まとめて（あるいは全ての内容を一度に）入出力することになる。

ファイル入出力を頻繁に実行しながら他の処理を並行して実行することが求められるアプリケーションを作成する場合に、この aiofiles が役立つ。

■ 非同期処理の通信への応用

asyncio による非同期処理の通信への応用に関しては、後の「5 TCP/IP による通信」(p.367) の中の「5.1.6 実用的な通信機能を実現する方法」(p.370) で解説する。

参考)

サードパーティが開発して公開している uvloop ライブラリを用いると高速なイベントループが実現できる。これは、JavaScript 処理系の V8 エンジンで採用されているイベントループ機能 (libuv) を Python 向けに応用したものである。

4.6 処理のスケジューリング

指定した時間が経過した後に関数を呼び出す、あるいは指定した時刻に関数を呼び出す方法について説明する。

4.6.1 sched モジュール

Python 処理系に標準的に提供されている sched モジュールを利用すると、指定した時間が経過した後にスケジュール登録しておいた関数を呼び出す（実行する）ことができる。このモジュールによるスケジューリングは、関数呼び出しのスケジュールを**スケジューラ**（scheduler）**オブジェクト**に対して登録し、それを実行するという手順となる。

このモジュールは使用に先立って

```
import sched
```

として処理系に読み込んでおく必要がある。

4.6.1.1 基本的な使用方法

スケジューラオブジェクトを作成するには下記のように scheduler コンストラクタを使用する。

```
s = sched.scheduler()
```

この例では、変数 s にスケジューラオブジェクトが得られている。

イベント（実行したい関数）は、スケジューラオブジェクトに対して enter メソッドを使用して登録する。

書き方： `enter(経過時間, 優先順位, 関数名, argument=関数に渡す引数のタプル)`

「経過時間」の単位は秒（浮動小数点数）である。「優先順位」は登録するイベントの優先順位であり、値の小さい方が優先度が高い。enter メソッドは実行後、**イベント（Event）オブジェクト**を返す。イベントオブジェクトは登録された個々のスケジュールを意味する。イベントの登録ができれば、その直後にスケジューラオブジェクトに対して run メソッドを実行する。これによってスケジュールの終了待ちの状態となる。

注） イベント実行までの経過時間は、enter メソッドでイベントを登録した時点が基準となる。

スケジューラオブジェクトには複数のイベントを登録することができる。

次に示すサンプルプログラム sched01.py は 3 つのイベントをスケジューリングする例である。

プログラム： sched01.py

```
1  # coding: utf-8
2  import sched
3  import time
4
5  # イベントとして実行する関数
6  def job( m ):
7      print( m )          # メッセージを表示するだけの関数
8
9  # イベントのスケジューリング
10 s = sched.scheduler()    # スケジューラの作成
11 e1 = s.enter(3,1,job,argument=('実行1',)) # イベント（3秒後）を登録
12 e2 = s.enter(4,1,job,argument=('実行2',)) # イベント（4秒後）を登録
13 e3 = s.enter(5,1,job,argument=('実行3',)) # イベント（5秒後）を登録
14 t1 = time.time()        # イベント登録完了時刻
15 print('イベントを登録しました。')
16 s.run()                  # 開始（イベント終了を待つ）
17 t2 = time.time()        # 全イベントが終了した時刻
18 print( t2-t1, '秒後に全てのイベントが終了しました。' )
```

プログラムの 6,7 行目に定義している関数は与えられたものをそのまま表示するもので、これをスケジューリング対象のイベントとしている。11～13 行目で関数 job を 3 秒後、4 秒後、5 秒後にそれぞれ実行するものとしてスケジューリング（登録）している。

このプログラムを実行すると次のような出力が得られる。

実行例. (スクリプトとして実行)

イベントを登録しました.

実行 1

実行 2

実行 3

5.015453100204468 秒後に全てのイベントが終了しました.

関数 job による出力が、設定された時間の後に表示されることを確認されたい.

4.6.1.2 イベントを独立したスレッドで実行する方法

スケジューラオブジェクトに登録された個々のイベントは全て同じスレッド (Thread) で実行される. 従って, 1 つのイベントが実行中である場合は, それが終了するまで次のイベントは実行されない. このことを次のサンプルプログラム sched02.py で示す.

プログラム: sched02.py

```
1 # coding: utf-8
2 import sched
3 import time
4
5 # イベントとして実行する関数
6 def job( m ):
7     for i in range(3): # 3秒間に渡ってメッセージを3回出力する
8         print(m)
9         time.sleep(1)
10
11 # イベントのスケジューリング
12 s = sched.scheduler() # スケジューラの作成
13 e1 = s.enter(4.00,1,job,argument=('イベント1',))
14 e2 = s.enter(4.33,1,job,argument=('イベント2',))
15 e3 = s.enter(4.66,1,job,argument=('イベント3',))
16 t1 = time.time() # イベント登録完了時刻
17 print('イベントを登録しました. ')
18 s.run() # 開始 (イベント終了を待つ)
19 t2 = time.time() # 全イベントが終了した時刻
20 print( 't2-t1, '秒後に全てのイベントが終了しました. ' )
```

このプログラムで定義されている関数 job は, 3 秒間に渡ってメッセージを 3 回出力するものであり, 一回の実行に約 3 秒かかる. 従って, 13~15 行目で登録された 3 つのイベントは全て 4 秒を過ぎた後に並行して実行されるのではなく, 1 つずつ順番に実行される. このことは次の実行例で確認できる.

実行例. (スクリプトとして実行)

イベントを登録しました.

イベント 1

イベント 1

イベント 1

イベント 2

イベント 2

イベント 2

イベント 3

イベント 3

イベント 3

13.054096221923828 秒後に全てのイベントが終了しました.

1 つのイベントの実行が終了した後, 次のイベントが実行されていることがわかる.

個々のイベントをそれぞれ独立した時間軸で実行する (それぞれが同時に実行されているように扱う) には, 各イベントで実行する関数を別々のスレッド²⁰⁴ にすると良い. このことに関してサンプルプログラム sched03.py で例示する.

プログラム: sched03.py

```
1 # coding: utf-8
2 import sched
3 import time
```

²⁰⁴ 「4.4.1 マルチスレッド」(p.255) 参照のこと.

```

4 import threading
5
6 # メッセージを出力する関数
7 def job( m ):
8     for i in range(3): # 3秒間に渡ってメッセージを3回出力する
9         print(m)
10        time.sleep(1)
11
12 # イベントとして実行する関数（上記の関数を独立したスレッドで実行する）
13 def job_th(m):
14     th = threading.Thread(target=job, args=(m,))
15     th.start()
16
17 # イベントのスケジューリング
18 s = sched.scheduler() # スケジューラの作成
19 e1 = s.enter(4.00,1,job_th,argument=('イベント1',))
20 e2 = s.enter(4.33,1,job_th,argument=('イベント2',))
21 e3 = s.enter(4.66,1,job_th,argument=('イベント3',))
22 t1 = time.time() # イベント登録完了時刻
23 print('イベントを登録しました. ')
24 s.run()
25 t2 = time.time() # runメソッドが終了した時刻
26 print( t2-t1, '''秒後にrunメソッドが終了しました.
27 プログラムを終了するには Enter を押してください. ''' )
28 input() # 入力待ち（Enterでプログラムが終了）

```

このプログラムで定義している関数 job は先の sched02.py と同じものであるが、その関数を独立したスレッドで実行するための別の関数 job_th が定義されており、これをイベントとしてスケジューリングしている。これにより、全てのイベントは 4 秒を過ぎた後に並行して実行されるように見える。このことが次の実行例で確認できる。

実行例。（スクリプトとして実行）

```

イベントを登録しました。
イベント 1
イベント 2
イベント 3
4.662025690078735 秒後に run メソッドが終了しました。
プログラムを終了するには Enter を押してください。
イベント 1
イベント 2
イベント 3
イベント 1
イベント 2
イベント 3
←ここで Enter を押すことでプログラムが終了する。

```

各スレッドにおける関数 job が独立したスレッドとして同時に実行されている様子がわかる。また、最後に登録したイベント e3 を実行し終えた（スレッド起動が完了した）時点で run メソッド（24 行目）が完了していることもわかる。

■ 工夫

更に、run メソッドを実行する処理を独立したスレッドにすると、run メソッドの終了まで待たされることがなくなる。これをサンプルプログラム sched04.py で例示する。

プログラム：sched04.py

```

1 # coding: utf-8
2 import sched
3 import time
4 import threading
5
6 # メッセージを出力する関数
7 def job( m ):
8     for i in range(3): # 3秒間に渡ってメッセージを3回出力する
9         print(m)
10        time.sleep(1)
11
12 # イベントとして実行する関数（上記の関数を独立したスレッドで実行する）
13 def job_th(m):
14     th = threading.Thread(target=job, args=(m,))
15     th.start()

```

```

16
17 # runメソッドを実行する関数
18 def mgr(s):
19     s.run()
20
21 # イベントのスケジューリング
22 s = sched.scheduler() # スケジューラの作成
23 e1 = s.enter(4.00,1,job_th,argument=('イベント1',))
24 e2 = s.enter(4.33,1,job_th,argument=('イベント2',))
25 e3 = s.enter(4.66,1,job_th,argument=('イベント3',))
26 t1 = time.time() # イベント登録完了時刻
27 print('イベントを登録しました. ')
28
29 sth = threading.Thread(target=mgr,args=(s,)) # 関数mgrを独立スレッドにして
30 sth.start() # それを起動する
31
32 t2 = time.time() # runメソッドが終了した時刻
33 print( t2-t1, '''秒後にrunメソッドが終了しました.
34 プログラムを終了するには Enter を押してください. ''' )
35 input() # 入力待ち (Enterでプログラムが終了)

```

このプログラムは先の sched03.py と似ているが、run メソッドの実行を関数 mgr として定義し、それを独立したスレッドとして実行 (29,30 行目) している。これにより、全てのイベント登録と run メソッドの実行が速やかに終了し、プログラムの実行位置が 32 行目に移る。このことが次の実行例で確認できる。

実行例. (スクリプトとして実行)

```

イベントを登録しました.
0.0 秒後に run メソッドが終了しました.
プログラムを終了するには Enter を押してください.
イベント 1
イベント 2
イベント 3
イベント 1
イベント 2
イベント 3
イベント 1
イベント 2
イベント 3
←ここで Enter を押すことでプログラムが終了する.

```

イベントの実行が終了するのを待たずにプログラムが最終行に至っていることがわかる。

4.6.1.3 イベントの管理

イベント管理に関する機能を表 38 に示す。

表 38: イベント管理のための機能 (スケジューラオブジェクト S に対するメソッド/プロパティ)

記述	解説
S.cancel(e)	S に登録されているイベント e をキャンセルする。
S.empty()	S に登録されているイベントがなければ True を、存在すれば False を返す。
S.queue	S に登録されているイベントのリスト。

4.6.2 schedule モジュール

Python システムに標準添付されているモジュール以外にも、処理のスケジューリングのためのオープンソースのモジュール schedule²⁰⁵ が公開されている。ここでは schedule モジュールの最も基本的な使用方法について解説する。

次のような関数 job の実行のスケジューリングを例に挙げて説明する。

²⁰⁵このモジュールは「`pip install schedule`」でインストールできる。詳しくは公式インターネットサイト <https://pypi.org/project/schedule/> を参照のこと。

例. 日付と時刻を付けてメッセージを出力する関数 job

```
>>> from datetime import datetime Enter    ← datetime モジュールの読み込み
>>> def job(m): Enter    ← 日付・時刻付きでメッセージを表示する関数
...     print( str(datetime.now())+'%t:'%str(m) ) Enter
... Enter    ← 関数定義の記述の終了
>>> job(' メッセージ') Enter    ← 実行
2020-10-14 14:16:28.811510 :メッセージ    ← 実行結果の出力
```

4.6.2.1 一定の時間間隔で処理を起動する方法

先に定義した関数 job を 1 分間隔で実行するには次のようにスケジューリングする。

例. 1 分間隔で実行を繰り返すスケジューリング (先の例の続き)

```
>>> import schedule Enter    ← schedule モジュールの読み込み
>>> e1 = schedule.every(1).minutes.do( job, '1 分間隔のジョブ' ) Enter    ← ジョブの登録
```

この結果, 関数 job を 1 分毎に実行するスケジューリングができる. every(1).minutes は「1 分毎」を意味する記述であり, それに続く do(...) で実行する関数をスケジュールする.

書き方: do(関数名, 関数に与える引数の並び...)

先の例の実行によってジョブ (job) オブジェクト (スケジュールされる処理の単位) e1 が得られている. 同様に, 20 秒間隔でジョブをスケジュールするには次のようにする.

例. 20 秒間隔で実行を繰り返すスケジューリング (先の例の続き)

```
>>> e2 = schedule.every(20).seconds.do( job, '20 秒間隔のジョブ' ) Enter
```

以上で 1 分毎, 20 秒毎のスケジューリングができたことになる. このスケジュールを実行するには次のような繰り返し処理を実行する.

例. スケジュールを実行するループ (先の例の続き)

```
>>> import time Enter    ← time モジュールの読み込み
>>> while True: Enter    ← ループ記述の開始
...     schedule.run_pending() Enter    ← スケジュールを実行する処理
...     time.sleep(1) Enter    ← 1 秒待って次の回に進む
... Enter    ← ループ記述の終了

2020-10-14 15:00:51.309569 :20 秒間隔のジョブ    ←以降, スケジュール実行の過程
2020-10-14 15:01:11.494809 :20 秒間隔のジョブ    ↓
2020-10-14 15:01:31.661763 :1 分間隔のジョブ    :
2020-10-14 15:01:31.661763 :20 秒間隔のジョブ
2020-10-14 15:01:51.819587 :20 秒間隔のジョブ
2020-10-14 15:02:11.991623 :20 秒間隔のジョブ
2020-10-14 15:02:32.180156 :1 分間隔のジョブ
2020-10-14 15:02:32.180156 :20 秒間隔のジョブ
```

これは, 1 秒毎にスケジュールを監視して実行するループである. この例にある run_pending メソッドがスケジュールを監視して実際にジョブを起動する. このように Python の対話モードでループを実行した場合, スケジュール実行の繰り返しを停止するには Ctrl + C とキー入力 (次の例) する.

例. Ctrl + C による停止時の表示 (先の例の続き)

```
Traceback (most recent call last):    ← 処理中断のメッセージ
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
```

do メソッドで登録したジョブはキャンセルされるまで有効であり, 再度スケジュールのループを実行した際, 起動の対象となる. ジョブのキャンセルには cancel.job メソッドを用いる.

例. ジョブのキャンセル (先の例の続き)

```
>>> schedule.cancel_job(e1)  ←ジョブ e1 のキャンセル
>>> schedule.cancel_job(e2)  ←ジョブ e2 のキャンセル
```

■ 参考

ジョブとしてスケジューリングする関数は、sched モジュールに関する解説の「4.6.1.2 イベントを独立したスレッドで実行する方法」(p.292) で説明したように、必要に応じて独立したスレッドにしておくが良い。

4.6.2.2 指定した時刻に処理を起動する方法

毎日、指定した時刻にジョブを起動する例を示す。

例. 毎日 15:34:00 にジョブを起動する例 (先の例の続き)

```
>>> e1 = schedule.every().day.at('15:34:00').do( job, '毎日 15:34:00 のジョブ' ) 
>>> while True:  ←ループ記述の開始
...     schedule.run_pending() 
...     time.sleep(1) 
...  ←ループ記述の終了
2020-10-14 15:34:00.782234 :毎日 15:34:00 のジョブ ←ジョブからの出力
```

この例のように `every().day.at(時刻)` という形でジョブ起動の時刻を設定する。

(この後 `Ctrl` + `C` でループを停止して `schedule.cancel_job(e1)` でジョブをキャンセルしたものとする)

留意事項：

上記の例を実行する際は実行環境の時刻を考え、ジョブ起動の時刻を適宜調整されたい。

例. 毎分の 00 秒, 20 秒, 40 秒にジョブを起動する例 (先の例の続き)

```
>>> e1 = schedule.every().minute.at(':00').do( job, '毎分 00 秒のジョブ' ) 
>>> e2 = schedule.every().minute.at(':20').do( job, '毎分 20 秒のジョブ' ) 
>>> e3 = schedule.every().minute.at(':40').do( job, '毎分 40 秒のジョブ' ) 
>>> while True:  ←ループ記述の開始
...     schedule.run_pending() 
...     time.sleep(1) 
...  ←ループ記述の終了
2020-10-14 16:09:20.925096 :毎分 20 秒のジョブ ←ジョブからの出力
2020-10-14 16:09:40.104823 :毎分 40 秒のジョブ ←ジョブからの出力
2020-10-14 16:10:00.223926 :毎分 00 秒のジョブ ←ジョブからの出力
2020-10-14 16:10:20.442045 :毎分 20 秒のジョブ ←ジョブからの出力
```

(この後 `Ctrl` + `C` でループを停止して

```
schedule.cancel_job(e1); schedule.cancel_job(e2); schedule.cancel_job(e3)
```

でジョブをキャンセルしたものとする)

留意事項：

上記の例を実行する際は実行環境の時刻の秒が 40~00 の間にあるタイミングで実行されたい。

例. 毎時の 18 分, 19 分にジョブを起動する例 (先の例の続き)

```
>>> e1 = schedule.every().hour.at(':18').do( job, '毎時 18 分のジョブ' ) 
>>> e2 = schedule.every().hour.at(':19').do( job, '毎時 19 分のジョブ' ) 
>>> while True:  ←ループ記述の開始
...     schedule.run_pending() 
...     time.sleep(1) 
...  ←ループ記述の終了
2020-10-14 16:18:00.353603 :毎時 18 分のジョブ ←ジョブからの出力
2020-10-14 16:19:00.795833 :毎時 19 分のジョブ ←ジョブからの出力
```

(この後 `Ctrl` + `C` でループを停止して

```
schedule.cancel_job(e1); schedule.cancel_job(e2)
```

でジョブをキャンセルしたものとする)

留意事項：

上記の例を実行する際は実行環境の時刻を考え、ジョブ起動の分を適宜調整されたい。

4.7 ジェネレータ

4.7.1 ジェネレータ関数

`range` 関数が生成する `range` オブジェクトのように、繰り返しの制御に与えるデータ列は必要に応じて生成されることが望ましい。特にサイズの大きなデータ列に対して繰り返しの制御を行う際には、システムの記憶資源の制限や処理の実行速度の面からデータ列は順次生成されることが望ましい。繰り返し制御のためにデータを順次生成するものとしてジェネレータ関数がある。

例として次のようなジェネレータ関数について考える。

例. 整数を際限なく生成するジェネレータ関数 `nbr`

```
>>> def nbr(): Enter ←ジェネレータ関数 nbr の定義の開始
...     n = 0 Enter
...     while True: Enter
...         yield n Enter
...         n += 1 Enter
... Enter ←ジェネレータ関数 nbr の定義の終了
>>> ←対話モードのプロンプトに戻った
```

これはジェネレータ関数の1つの例であるが、無限に整数列を生成する形になっていることがわかる。通常関数定義の場合は、これはいわゆる無限ループであり、特に忌避されるものである。ただし、定義の中に `yield` という文があることが重要である。

`yield` 文は、その関数が生成するデータを意味するものであり、この関数は `for` などの制御構文から呼び出される毎に、`yield` 文のところで呼び出し元のプログラムと同期すると解釈することができる。

イテレータに対してするように `next` 関数を用いてこの関数 `nbr` を呼び出す例を次に示す。

例. `next` 関数で値を順に取り出す (先の例の続き)

```
>>> q = nbr() Enter ←関数 nbr をジェネレータとして q に与える
>>> q Enter ←内容確認
<generator object nbr at 0x000001600A41C5C8> ←ジェネレータとなっていることがわかる
>>> next(q) Enter ←next 関数による値の取り出し
0 ←取り出し結果
>>> next(q) Enter ←更に次の値の取り出し
1 ←取り出し結果
>>> next(q) Enter ←更に次の値の取り出し
2 ←取り出し結果
```

この例から、無限ループによる制御不能の状態に陥ることなく、関数 `nbr` から値を際限なく取り出せることがわかる。

`yield` 文は `return` 文のように値を返すものであるが、呼び出し元と同期した形で次々と列の要素として値を生成するものと解釈する。

戻り値の指定に `yield` 文を用いた関数はジェネレータ関数となる。ジェネレータ関数において `return` 文は使用できない。(無視される)

4.7.2 ジェネレータ式

「2.6.1.7 for を使ったデータ構造の生成（要素の内包表記）」(p.92) のところでデータ列の内包表記について説明したが、タプルに対する内包表記はジェネレータ式となる。(次の例参照)

例. ジェネレータ式

```
>>> g = ( 2*x for x in range(4) )  [Enter]    ←ジェネレータ式を生成
>>> g  [Enter]    ←内容確認
<generator object <genexpr> at 0x000001600A34E7D8>    ←ジェネレータとなっていることがわかる
>>> for m in g:  [Enter]    ←個々に値を取り出してみる
...     print( m )  [Enter]
...  [Enter]    ←繰り返しの終了
0      ←結果表示（ここから）
2
4
6      ←結果表示（ここまで）
```

これは、リスト [0,1,2,3] の各要素を2倍して表示したと見ることができるが、g に生成されたジェネレータ式は参照された時点で値を生成する。長大なリストを繰り返し制御に与えるよりも、ジェネレータ式的应用により記憶資源と実行速度の面で効率を高めることができる場合がある。

4.7.3 ジェネレータの入れ子

ジェネレータとして値を返す yield の後に「from イテラブル」を記述することで、与えたイテラブルの要素を順に返すことができる。

例. yield from の記述例

```
>>> def g1():  [Enter]    ←ジェネレータ関数 g1 の定義の開始
...     yield from ['d','e','f']  [Enter]    ←リストの要素を順に返す記述
...  [Enter]    ←ジェネレータ関数 g1 の定義の終了
```

この g1 から順に値を取り出す例を次に示す。

例. g1 から値を取り出す（先の例の続き）

```
>>> for x in g1():  print(x,end=' ')  [Enter]    ← g1 から順に値を取り出す処理
... else:  print()  [Enter]
...  [Enter]
d e f      ←実行結果
```

yield from の後に別のジェネレータ関数を与えることもできる。

例. ジェネレータの入れ子（先の例の続き）

```
>>> def g2():  [Enter]    ← g1 を呼び出す g2
...     yield from 'abc'  [Enter]
...     yield from g1()  [Enter]
...  [Enter]    ← g2 の定義の終了
>>> for x in g2():  print(x,end=' ')  [Enter]    ← g2 から順に値を取り出す処理
... else:  print()  [Enter]
...  [Enter]
a b c d e f      ←実行結果
```

4.8 モジュール、パッケージの作成による分割プログラミング

実用的なアプリケーションプログラムの開発においては、多くの関数やクラスを定義する。また、システムの機能を細分化して複数のソースプログラムとしてアプリケーションを作り上げることが一般的である。更に、汎用性の高い関数やクラスは、別のアプリケーションを開発する際にも再利用できることが望ましい。

これまで、様々なモジュールやパッケージの利用方法について説明したが、それらは多くの開発者（サードパーティー）によって開発されたプログラムであり、汎用性の高さゆえに公開され広く利用されている。再利用が望まれる関数やクラスはこのように**モジュールやパッケージ**という形で用意しておくが、ここでは、独自のモジュールやパッケージを作成する方法について説明する。

4.8.1 モジュール

4.8.1.1 単体のソースファイルとしてのモジュール

最も簡単な方法として、1つのソースファイルに関数やクラスの定義を記述してモジュールを作成するという方法があるが、これに関してサンプルを示しながら説明する。

次のようなプログラム（モジュールファイル）MyModule.pyを用意する。

プログラム：MyModule.py

```
1 # coding: utf-8
2 # 加算関数
3 def kasan(x,y):
4     print('module:',__name__)
5     return x + y
6
7 # 乗算関数
8 def jouzan(x,y):
9     print('module:',__name__)
10    return x * y
```

この場合のモジュールの名前はそのファイル名（拡張子は除く）であり、このモジュールには2つの関数 kasan と jouzan が定義されている。このモジュールは他のプログラムやPython インタプリタに読み込んで（import して）使用することができる。

■ モジュールのパスについて

Python 言語処理系は import 対象のモジュールを sys.path に登録されているディレクトリの中から探す。sys.path はリストであり、予めいくつかのディレクトリが登録されているが、Python 利用者が独自に作成したモジュールを import の対象とするには、そのモジュールの属するディレクトリが sys.path に含まれているかを確認すること。含まれていなければ、そのモジュールが属するディレクトリのパスを sys.path に追加しなければならない²⁰⁶。これに関しては巻末付録の「D.1.3 ライブラリのパス：sys.path」（p.434）で解説する。

上の MyModule.py がモジュールとして import できる状態での使用例を次に示す。

例. Python インタプリタからモジュール MyModule.py を利用する例 (1)

```
>>> import MyModule  Enter  ←読み込み
>>> MyModule.kasan( 2, 3 )  Enter  ← kasan の実行
module:  MyModule          ←モジュール名の表示
5                             ←処理結果
```

この例のように、**モジュール名.関数** として²⁰⁷ モジュールに定義された関数を呼び出すことができる。

Python では大域変数（グローバル変数）__name__が定義されており、現在実行されているモジュール名がそこに保持されている。先のプログラム MyModule.py では kasan, jouzan 関数が自身が属するモジュールの名前を出力する形になっており、実行結果としてモジュール名である MyModule を出力している。

メインプログラムのモジュール名は '__main__' である。（詳しくは後の「4.8.2.1 モジュールの実行」（p.301）を参照のこと）

²⁰⁶JupyterNotebook など一部の対話環境では、sys.path に変更を加えた後、セッションの再起動が必要となる場合もあるので注意すること。

²⁰⁷詳しくは巻末付録「D.1 ライブラリの読み込みに関すること」（p.433）を参照のこと。

例. 対話モードでのモジュール名の確認

```
>>> print(__name__) Enter ←モジュール名の表示 (メイン)
__main__ ←モジュール名の表示
```

モジュールを使用した後は、モジュールファイルと同じディレクトリ内にサブディレクトリ `__pycache__` が作成される。

関数呼び出し時にモジュール名を省略することもできる。(次の例参照)

例. Python インタプリタからモジュール `MyModule.py` を利用する例 (2)

```
>>> from MyModule import kasan, jouzan Enter ←読み込み
>>> kasan(2,3) Enter ← kasan の実行
module: MyModule ←モジュール名の表示
5 ←処理結果
>>> jouzan(3,4) Enter ← jouzan の実行
module: MyModule ←モジュール名の表示
12 ←処理結果
```

この例にあるように、

from モジュール名 import 使用する関数 (クラス) 名

として²⁰⁸ モジュールを読み込む。

規模の大きなプログラムを開発する際は、複数のサブディレクトリと複数のファイルから構成されるパッケージの形にするのが一般的である。

4.8.2 パッケージ (ディレクトリとして構成するライブラリ)

複数のソースファイルから構成されるモジュール群をパッケージという。1つのパッケージは1つのディレクトリとして作成する。またそのディレクトリ内に初期化スクリプト `__init__.py` を作成しておく²⁰⁹。以下に例を挙げてパッケージ作成の方法について解説する。

注意) パッケージを `import` 可能にするには、モジュールの場合と同様に `sys.path` に注意すること。

■ パッケージ作成の例

次のようなディレクトリ構成のパッケージについて考える。

./MyPackage	(ディレクトリ)
├-- __init__.py	(初期化スクリプト)
├-- kasan.py	(テキスト形式のソースプログラム)
├-- jouzan.py	(テキスト形式のソースプログラム)
└-- cdir.py	(テキスト形式のソースプログラム)

この場合のパッケージ名は、パッケージのファイル (フォルダ) を含む最上位のディレクトリ名である。この例では `MyPackage` がパッケージ名となる。次に `kasan.py` と `jouzan.py` の内容を示す。

プログラム: kasan.py

```
1 # coding: utf-8
2 # 加算関数
3 def kasan(x,y):
4     print('module:',__name__)
5     return x + y
6
7 # テスト実行用メイン部
8 if __name__ == '__main__':
9     print('テスト実行: kasan(2,3)=', kasan(2,3))
```

²⁰⁸ 詳しくは巻末付録「D.1 ライブラリの読み込みに関すること」(p.433)を参照のこと。

²⁰⁹ 初期化のための処理を特に記述しない場合は `__init__.py` の内容は空でも良い。

プログラム：jouzan.py

```
1 # coding: utf-8
2 # 乗算関数
3 def jouzan(x,y):
4     print('module:',__name__)
5     return x * y
6
7 # テスト実行用メイン部
8 if __name__ == '__main__':
9     print('テスト実行：jouzan(3,4)=',jouzan(3,4))
```

(cdir.py については後で触れる)

これらパッケージを読み込んで使用する例を次に示す。

例. パッケージを読み込んで使用する

```
>>> from MyPackage.kasan import kasan  Enter    ←パッケージの読み込み (1)
>>> from MyPackage.jouzan import jouzan  Enter    ←パッケージの読み込み (2)
>>> kasan(2,3)  Enter    ← kasan の実行
module:  MyPackage.kasan  ←モジュール名の表示
5                      ←処理結果
>>> jouzan(3,4)  Enter    ← jouzan の実行
module:  MyPackage.jouzan  ←モジュール名の表示
12                      ←処理結果
```

この例にあるように、

from パッケージ名. モジュールファイル名 import 使用する関数 (クラス) 名

として²¹⁰ モジュールを読み込む。ここで言う**モジュールファイル**は、パッケージのディレクトリ配下にあるソースファイル（拡張子は除く）のことである。

この例の2つのソースプログラムの7~9行目には「テスト実行用**メイン部**」というものが記述されているが、これについては次に説明する。

4.8.2.1 モジュールの実行

モジュールは他のプログラム（メインプログラム）から呼び出して使用するものであるが、開発中にそのモジュールをテストするために、テスト実行用のメインプログラムを別のソースファイルとして作成するのは煩わしい場合がある。この問題を軽減するために、モジュールのファイルの中に、テスト実行用の**メイン部**を記述しておいて、直接そのモジュールをPython インタプリタで実行するのが良い。例えば先のパッケージの例において、次のようにしてOSのコマンドシェルからモジュールを直接実行することができる。

Windows での例

```
C:\¥Users¥katsu > py MyPackage/kasan.py  Enter    ← OS のシェルからモジュールを直接実行
module:  __main__    ←メインプログラムとして実行されていることがわかる
テスト実行：kasan(2,3)= 5    ←実行結果
```

Python のプログラミングにおいては、このような形（下記のような形）で明に**メイン部**を記述するスタイルが一般的である。

Python のプログラミングスタイル

```
if __name__ == '__main__':
    :
    (メインプログラム)
    :
```

4.8.3 モジュールが配置されているディレクトリの調査

作成したプログラムをパッケージとして1つのディレクトリにまとめる際に、パッケージのモジュールプログラム（スクリプト）で使用するデータ類を同一のディレクトリに含めることがある。この場合、スクリプトがデータにアク

²¹⁰詳しくは巻末付録「D.1 ライブラリの読み込みに関すること」(p.433)を参照のこと。

セスするために当該ディレクトリの位置（パス）を知る必要がある。モジュールの処理（関数やメソッド）の中で、当該スクリプトが配置されているディレクトリのパスを知るには「2.7.7.8 実行中のスクリプトに関する情報」（p.122）の所で紹介した方法を応用する。

実行中のスクリプトが自身のファイル名を取得するには、グローバル変数‘__file__’を参照する。更に、os.path モジュールの abspath 関数によって、‘__file__’の絶対パスを取得することができる。また、os.path モジュールの dirname 関数によって、それを格納するディレクトリ名を取得することができる。

以上のことを応用して次のようなモジュールファイル cdir.py をパッケージのディレクトリ MyPackage に作成する。

プログラム：cdir.py

```
1 # coding: utf-8
2 import os
3
4 # ディレクトリの調査
5 def chkPath():
6     p = os.path.dirname( os.path.abspath(__file__) )
7     return p
8
9 # テスト実行用メイン部
10 if __name__ == '__main__':
11     print( '__file__:', __file__ )
12     print( '格納場所:', chkPath() )
```

このモジュールの chkPath 関数が当該モジュールを格納するパスを返す。動作確認をするために、カレントディレクトリをパッケージのディレクトリ MyPackage にして cdir.py の実行を試みる。（下記）

例. cdir.py を直接実行する（Windows での例）

```
C:¥Users¥katsu¥MyPackage> py cdir.py      Enter      ←モジュールを実行
__file__:  C:¥Users¥katsu¥MyPackage¥cdir.py      ←モジュールファイルのパス
絶対パス:  C:¥Users¥katsu¥MyPackage      ←格納位置のパス
```

次にカレントディレクトリを別のパス（MyPackage の親ディレクトリ）にして、cdir.py をモジュールとして読み込み、chkPath 関数の実行を試みる。（下記）

例. モジュールとして chkPath を実行

```
>>> import MyPackage.cdir      Enter      ←モジュールの読み込み
>>> print( MyPackage.cdir.chkPath() )      Enter      ← chkPath 関数の実行
C:¥Users¥katsu¥MyPackage      ←結果表示
```

モジュールファイル cdir.py 自身のディレクトリが表示されている。

4.8.4 __init__.py について

パッケージのディレクトリ内に配置する __init__.py には、Python 処理系が当該パッケージを読み込む際に実行する処理（初期化などに関する処理）を記述する。具体的な内容としては、パッケージの開発者が自由に記述することができるが、ここでは応用例を1つ示す。

実際にパッケージを開発する際は、それを収めるディレクトリ内に複数のモジュールファイルを配置する。また、サブディレクトリを設置して「サブパッケージ」として階層的にモジュールファイル群を配置することもある。そのようなパッケージを利用する場合、

from パッケージ.サブパッケージ.サブサブパッケージ... import クラスや関数

などとして読み込むことになるが、このような煩雑な記述を簡略化することができる。先に示したパッケージの例 MyPackage において __init__.py の内容に次のように記述する。

プログラム：__init__.py

```
1 # coding: utf-8
2 # モジュールの読み込み
3 from MyPackage.kasan import kasan
4 from MyPackage.jouzan import jouzan
```

```
5 | from MyPackage.cdir import chkPath
```

これにより MyPackage を読み込む際に __init__.py の内容が実行され、それぞれのモジュールファイル内に記述された関数を 'MyPackage.' の接頭辞により呼び出すことができる。(次の例参照)

例. 簡略化されたパッケージの扱い

```
>>> import MyPackage Enter ←パッケージの読み込み
>>> MyPackage.kasan(2,3) Enter ← kasan の実行
module: MyPackage.kasan ←実行結果
5 ←実行結果
>>> MyPackage.jouzan(3,4) Enter ← jouzan の実行
module: MyPackage.jouzan ←実行結果
12 ←実行結果
>>> print( MyPackage.chkPath() ) Enter ← chkpath の実行
C:\Users\katsu\MyPackage ←実行結果
```

パッケージ／モジュールの読み込み方などに関しては、巻末付録「D.1 ライブラリの読み込みに関すること」(p.433)を参照のこと。

4.9 ファイル内でのランダムアクセス

通常の場合、ファイル入出力は順次アクセス（Sequential Access）と呼ばれる手法で行われる。すなわち、読み込みの際は先頭からファイル末尾（EOF: End Of File）に向けて順番に読み取られ、書き込みの際はその時点でのファイル末尾にデータが追加される。これに対して、記録媒体の任意のバイト位置にアクセスする手法をランダムアクセス（Random Access）という。具体的には、記録媒体の指定したバイト位置から内容を読み取ったり、指定したバイト位置にデータを書き込む手法を意味する。ここでは、ファイルに対するランダムアクセスのための基本的な事柄について説明する。

4.9.1 ファイルのアクセス位置の指定（ファイルのシーク）

通常の順次アクセスにおいては、現在開いているファイルのアクセス位置は「次にアクセスすべき位置」が自動的に取られるが、seek メソッドを使用すると、指定した任意のバイト位置にアクセス位置を移動することができる。seek メソッドの基本的な使い方を表 39 に示す。

表 39: ファイル f に対する seek メソッドの基本的な使い方

書き方	説明
f.seek(n)	ファイルの先頭からの相対位置（n バイト目：0 以上）にアクセス位置を移動する。
f.seek(n,0)	同上
f.seek(n,1)	現在のアクセス位置からの相対位置（n バイト目：負の値も可）にアクセス位置を移動する。
f.seek(n,2)	ファイルの末尾からの相対位置（n バイト目：0 以下）にアクセス位置を移動する。

※ seek メソッド実行後はシーク結果のバイト位置（ファイルの先頭を基準とする）の値を返す。

4.9.2 サンプルプログラム

ファイルへのランダムアクセスを行うサンプルプログラム fram01.py を示す。このプログラムは 10 バイトのレコードを 3 件持つファイルに対するランダムアクセスを行う。

プログラム：fram01.py

```
1  # coding: utf-8
2  #####
3  # ファイル内容を表示する関数 #
4  #####
5  def dumpf(f):
6      for i in range(3):
7          f.seek(i*10)
8          rec = f.read(10).decode('utf-8')
9          print(rec)
10         print('')
11
12 #####
13 # ファイルのランダムアクセスの試み #
14 #####
15 # ファイルの作成（空ファイルの準備）
16 f = open('fram01.dat', 'wb')
17 f.close()
18
19 # 読み書き可能な形で再度開く
20 f = open('fram01.dat', 'rb+')
21
22 #==== '0000:*****' の形（長さ10バイト）で3件書き込み ====
23 for i in range(3):
24     f.write('0000:*****'.encode('utf-8'))
25 # 内容確認
26 dumpf(f)
27
28 #==== 2レコード目に '0002: two' を書き込み ====
29 f.seek(20)
30 f.write('0002: two'.encode('utf-8'))
31 # 内容確認
32 dumpf(f)
33
```

```

34 #==== 0レコード目に '000z: zero' を書き込み ====
35 f.seek(0)
36 f.write( '000z: zero'.encode('utf-8') )
37 # 内容確認
38 dumpf(f)
39
40 #==== 1レコード目に '0001: one' を書き込み ====
41 f.seek(10)
42 f.write( '0001: one'.encode('utf-8') )
43 # 内容確認
44 dumpf(f)
45
46 # ファイルサイズの取得
47 p = f.seek(0,2)
48 print('File size:',p,'bytes')
49
50 #=== 終了 ===
51 f.close()

```

解説：

16,17 行目で空のファイルを新規に作成して、20～24 行目でファイル内容を初期化している。この結果、ファイルの内容が '0000:*****' というレコードを 3 つ持つ形となる。

5～10 行目はファイルの内容全体を表示するための関数（名前：dumpf）で、プログラム内で度々呼び出されている。

29～44 行目ではファイル内のバイト位置を seek メソッドで直接指定してランダムアクセスしている。（順不同の書き込み）

このプログラムを実行した結果を次に示す。

```

0000:*****   ←ファイル内容の初期化の結果
0000:*****
0000:*****

0000:*****
0000:*****
0002:  two    ← 2 レコード目に書き込み

000z:  zero   ← 0 レコード目に書き込み
0000:*****
0002:  two

000z:  zero
0001:  one    ← 1 レコード目に書き込み
0002:  two

File size: 30 bytes

```

4.10 オブジェクトの保存と読み込み：pickle モジュール

Python 処理系に標準的に添付されている pickle モジュールを利用すると、オブジェクト（データ構造）をバイト列²¹¹（バイナリ形式のデータ列）に変換（シリアライズ）する、あるいはその逆の変換を行うことができる。このモジュールが提供する代表的な関数を表 40 に挙げる。

表 40: pickle モジュールが提供する代表的な関数

関数	説明
<code>dumps(オブジェクト)</code>	「オブジェクト」をバイト列に変換したものを返す。
<code>dump(オブジェクト, ファイル)</code>	「オブジェクト」をバイト列に変換して「ファイル」に保存する。
<code>loads(バイト列)</code>	「バイト列」を展開して元のオブジェクトにして返す。
<code>load(ファイル)</code>	<code>dump</code> 関数により保存されたファイルを読み込み、元のオブジェクトにして返す。

例. リスト⇄バイト列の変換

```
>>> import pickle  ←モジュールの読み込み
>>> d = [1,2,3,4,5,6,7,8,9,10]  ←リストの用意
>>> bn = pickle.dumps(d)  ←バイト列に変換
>>> bn  ←内容確認
b'\x80\x03q\x00(K\x01K\x02K\x03K\x04K\x05K\x06K\x07K\x08K\tK\n.'  ←バイト列になっている
>>> pickle.loads(bn)  ←リストに戻す
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  ←元のリストになっている
```

`dumps`, `dump` 関数によって得られたバイト列はデータ型に関する情報などを含んでおり、元のデータに復元することができる。

次に、定義されたクラスのインスタンスとして生成されたオブジェクトをファイルに保存し、それを再び読み込んで元のオブジェクトに復元する例を示す。次に示すサンプルプログラム `pickle02class.py` は三角関数の表を生成してプロットするためのクラス `Data` を定義している。（グラフの描画には `matplotlib` ライブラリ²¹² を使用する）

プログラム：pickle02class.py

```
1  # coding: utf-8
2  import math                # 数学関数のモジュール
3  import matplotlib.pyplot as plt  # グラフ描画ライブラリ
4
5  #--- クラス定義 ---
6  class Data: # 三角関数の表を扱うクラス
7      #--- コンストラクタ ---
8      def __init__(self):
9          self.tri = dict()  # 辞書として空の表を生成
10     #--- データの生成 ---
11     def gen(self):
12         for ix in range(629):
13             x = ix / 100.0    # 定義域
14             ysin = math.sin(x); ycos = math.cos(x)    # 正弦関数, 余弦関数
15             self.tri[x] = [round(ysin,3),round(ycos,3)] # を生成してリストにする
16     #--- データ列の取り出し ---
17     def qx(self):
18         return list( self.tri.keys() )
19     def qsin(self):
20         return [self.tri[x][0] for x in self.qx()]
21     def qcos(self):
22         return [self.tri[x][1] for x in self.qx()]
23     #--- グラフのプロット ---
24     def plotTriangle(self):
25         plt.figure( figsize=(6,2.8) )
```

²¹¹ 「2.7.3.4 バイト列の扱い」(p.112) 参照のこと。

²¹² Python でグラフを描画する場合によく用いられるライブラリ（公式インターネットサイト：<https://matplotlib.org/>）拙書「Python3 ライブラリブック」でも基本的な使用方法を解説しています。

```

26         plt.plot( self.qx(), self.qsin() );      plt.plot( self.qx(), self.qcos() )
27         plt.show()

```

Data クラスは三角関数表を辞書オブジェクト `tri` として保持するもので、インスタンス生成後に `gen` メソッドによりデータを生成する。また、`plotTriangle` メソッドによりグラフを描画する。このクラスを用いて、

- 1) データの生成、グラフの描画、データの保存
- 2) 保存されたデータの復元 (Data クラス)、グラフの描画

の処理を行う例 (サンプルプログラム `pickle02-1.py`, `pickle02-2.py`) を示す。

プログラム：pickle02-1.py (データの生成、グラフ描画、データの保存)

```

1  # coding: utf-8
2  import pickle
3
4  #--- クラス定義読み込み ---
5  from pickle02class import Data
6
7  #--- データの生成とプロット ---
8  d = Data()          # オブジェクトの生成
9  d.gen()             # データの生成
10 d.plotTriangle()    # グラフのプロット
11
12 #--- pickleによるデータ保存 ---
13 f = open('pickle02.dat','wb')  # バイナリファイルを作成
14 pickle.dump( d, f )          # 保存
15 f.close()

```

プログラム：pickle02-2.py (データの読み込み、グラフの描画)

```

1  # coding: utf-8
2  import pickle
3
4  #--- pickleによるデータ読み込み ---
5  f = open('pickle02.dat','rb')  # バイナリファイルとして開く
6  d = pickle.load( f )          # 読み込み
7  f.close()
8
9  #--- グラフのプロット ---
10 d.plotTriangle()

```

これらプログラムを実行すると、Data クラスのオブジェクトがバイナリファイル `pickle02.dat` として保存される。特に重要な点として、読み込み側のプログラム `pickle02-2.py` においてクラスの定義の記述が無いことが挙げられる。pickle モジュールの機能によって保存された `pickle02.dat` にはデータのクラス定義の関連情報が含まれており^注、グラフ描画のための `plotTriangle` メソッドが実行できることが確認される。

両方のプログラムの実行において図 36 のようなグラフが表示される。

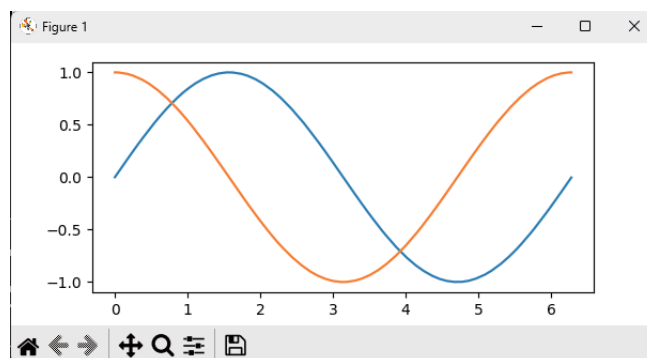


図 36: `pickle02-1.py`, `pickle02-2.py` で描画されるグラフ

注意)

先の例では、クラス Data の定義本体は pickle02.dat には含まれておらず、pickle02class.py の内容を参照している。具体的には、保存対象のオブジェクトをシリアル化されたバイト列データの中に、そのオブジェクトのクラス名や、そのクラスがどのモジュール（ファイル）に保存されているかの関連情報が保持されている。従って、Python 言語処理系が必要なモジュールファイルを見つけることができない場合はエラー（ModuleNotFoundError）が起こる。例えば先の例において、pickle02class.py がモジュールとして参照できない状況にあるとこの問題が起こる。

4.10.1 シリアル化と復元のカスタマイズ方法

Python のオブジェクトを dump, dumps でシリアル化し、load, loads で復元する処理において、通常はその処理過程をユーザが意識することはない。しかし、Python 3.10 以前の版では、スロットベースのクラス²¹³ のインスタンスをこの方法で処理することができず、シリアル化と復元の過程をユーザがカスタマイズする必要があった²¹⁴。また現在の版の Python においても、特殊な形式のオブジェクトを pickle で扱う場合は、処理過程をユーザが独自にカスタマイズしなければならないことがある。

次に示すサンプル pickle03class.py では、スロットベースのクラス Pos を定義している。このクラスは、2次元座標上の点を x, y の属性で現し、その原点 (0,0) からの距離を属性 _distance で表すものである。またこのクラスは、そのインスタンスをシリアル化と復元する際の処理のカスタマイズをしている。具体的には、クラス Pos に特殊メソッド __getstate__, __setstate__, __reduce__ を実装してカスタマイズしている。

プログラム：pickle03class.py（クラス定義）

```
1 class Pos:
2     __slots__ = ('x', 'y', '_distance')      # スロットとして属性を限定
3     # コンストラクタ
4     def __init__(self, px, py):
5         print('__init__: コンストラクタの実行')
6         self.x = px
7         self.y = py
8         self._distance = (px**2 + py**2)**0.5
9     # シリアル化の際の状態取得：すべての属性情報を辞書にして返す
10    def __getstate__(self):
11        state = {'x': self.x, 'y': self.y, '_distance': self._distance}
12        print('__getstate__: 保存のための状態取得 :', state)
13        return state
14    # 復元時のインスタンスの再構成：状態辞書から属性を再構成
15    def __setstate__(self, state):
16        print('__setstate__: 復元に用いる状態辞書 :', state)
17        self.x = state['x']
18        self.y = state['y']
19        self._distance = state['_distance']
20    # シリアル化の際に埋め込む情報：コンストラクタとその引数、状態辞書
21    def __reduce__(self):
22        print('__reduce__が起動')
23        return (self.__class__, (self.x, self.y), self.__getstate__())
24    # 状態表示
25    def show(self):
26        print(f' show: {self.x=}, {self.y=}, distance={self._distance}')

```

このクラス定義のカスタマイズの要点を以下に説明する。

■ シリアル化の対象となるオブジェクトの属性情報の収集：__getstate__ メソッド

シリアル化の対象となるオブジェクトの属性とその値の組を辞書の形で返す処理として実装する。

■ 復元処理：__setstate__ メソッド

属性とその値の組の辞書から実際にインスタンスの状態を構成する処理を実装する。

■ シリアル化に必要な機能とデータの収集：__reduce__ メソッド

__reduce__ メソッドは、オブジェクトを実際にシリアル化する際に呼び出されるもので、復元時にオブジェク

²¹³ 巻末付録「H スロットベースのクラス」(p.449) で解説する。

²¹⁴ Python 3.11 からはその必要がなくなった。

トを再構成するための関数（通常は当該クラスのコンストラクタ）と、属性情報辞書（通常は`__getstate__`で生成）を収集する。

`__reduce__` メソッドは、再構成用の関数、それに渡すための引数並びのタプル、属性情報辞書をタプルにして返す形にする。

`pickle03class.py` に記述されている `Pos` クラスでは、上記の特殊メソッド `__getstate__`、`__setstate__`、`__reduce__` やコンストラクタが呼び出されたことを `print` 関数で知らせる出力をしている。`Pos` クラスのインスタンスをシリアル化処理、復元する処理を次の `pickle03.py` で実行する。

プログラム：`pickle03.py`（シリアル化と復元）

```
1 import pickle
2 from pickle03class import Pos
3
4 print('■ インスタンス生成')
5 p1 = Pos(3,4)    # インスタンス生成
6 p1.show()        # 状態確認
7
8 f = open('pickle03.dat','wb')
9 print('■ シリアル化して保存します')
10 pickle.dump(p1,f)
11 f.close()
12
13 f = open('pickle03.dat','rb')
14 print('■ 復元します')
15 p2 = pickle.load(f)
16 f.close()
17
18 p2.show()
```

このプログラムを実行した際の出力の例を次に示す。

実行例. ターミナルのコマンドラインで実行（Python 3.13）

```
■ インスタンス生成
__init__: コンストラクタの実行
show: self.x=3, self.y=4, distance=5.0
■ シリアル化して保存します
__reduce__が起動
__getstate__: 保存のための状態取得: {'x': 3, 'y': 4, '_distance': 5.0}
■ 復元します
__init__: コンストラクタの実行
__setstate__: 復元に用いる状態辞書: {'x': 3, 'y': 4, '_distance': 5.0}
show: self.x=3, self.y=4, distance=5.0
```

処理の流れ、各メソッドが実行されたタイミングが確認できる。

4.10.2 シリアル化されたバイナリデータのプロトコルレベル

`pickle` の機能は Python の版の進展に従ってアップデートされてきており、オブジェクトのシリアル化の手法もアップデートされてきている。過去の `pickle` によって作成されたバイナリデータには、作成当時の手法（**`pickle` のプロトコル**：表 41）の情報が含まれており、新しい版の `pickle` はその情報（**プロトコルレベル**）を読み取って、復元処理を行う。

表 41: `pickle` のプロトコルレベル

プロトコルレベル	対応する Python の版	主な特徴
0	全ての版	ASCII 形式（テキスト）
1	全ての版	最初のバイナリ形式（後方互換性のために存在）
2	Python 2.3 以降	新スタイルクラスなどの効率化
3	Python 3.0 以降	bytes 型対応など（これ以降 Python 2 には非対応）
4	Python 3.4 以降	大容量への対応と高速化
5	Python 3.8 以降	アウト-オブ-バンドバッファなどの新機能

シリアル化処理の際、デフォルトでは処理系における最新のプロトコルレベルが採用される（例外もある）が、詳しくは次に説明する方法で調べると良い。

4.10.2.1 シリアライズされたバイナリデータのプロトコルレベルを調べる方法

標準ライブラリである `pickletools` が提供する `genops` 関数を用いることで、シリアル化されたバイナリデータを解析することができる。この関数はシリアル化されたバイナリデータのファイルから次々と構成要素を読み取るもので、「オペレーションコード、引数、要素の位置」を組みを返すイテレータとなる。

「オペレーションコード」の name 属性が 'PROTO' である場合の「引数」が、当該ファイルのプロトコルレベルである。また、古いプロトコルレベルのファイルはこの情報を持っていないこともある。このことを応用して、バイナリデータファイルのプロトコルレベルを調べる関数を実装したモジュールファイルを `pickleProtocol.py` に示す。

プログラム：pickleProtocol.py

```

1 import pickletools
2
3 def pickleProtocol(path):
4     with open(path, 'rb') as f:
5         for op, arg, _ in pickletools.genops(f):
6             if op.name == 'PROTO':
7                 return arg
8     return None # プロトコル0/1 (PROTOがない場合) はNone

```

このモジュールを用いて、先の pickle03.py が作ったバイナリデータファイル pickle03.dat を解析する例を次に示す。

例. pickle03.dat のプロトコルレベルを調べる

```
>>> from pickleProtocol import pickleProtocol Enter ←上記モジュールの読み込み
>>> pickleProtocol('pickle03.dat') Enter ←ファイル pickle03.dat のプロトコルレベルを調べる
4 ←結果
```

また、OS のコマンドシェルでも pickletools を実行して解析処理が行える。(次の例)

例. コマンド操作でバイナリファイルの情報を解析 (Windows 環境の PSF 版 Python の場合)

```
C:\Users\katsu\Python> py -m pickletools pickle03.dat  ←解析開始
0:  \x80 PROTO      4                ←プロトコルレベルの情報
2:  \x95 FRAME      71
11: \x8c SHORT_BINUNICODE 'pickle03class'
      :
      (以下省略)
```

4.10.2.2 シリアライズ時のプロトコルレベルの指定

dump, dumps に引数 'protocol=' を与えることで、シリアライズのプロトコルレベルを指定することができる。

例. プロトコルレベルを指定したシリアルライズ

```
>>> import pickle      Enter      ←モジュールの読み込み
>>> s0 = pickle.dumps( list(range(100000)), protocol=0 )      Enter      ←プロトコルレベル 0 を指定
>>> s1 = pickle.dumps( list(range(100000)), protocol=1 )      Enter      ←プロトコルレベル 1 を指定
>>> s2 = pickle.dumps( list(range(100000)), protocol=2 )      Enter      ←プロトコルレベル 2 を指定
>>> s3 = pickle.dumps( list(range(100000)), protocol=3 )      Enter      ←プロトコルレベル 3 を指定
>>> s4 = pickle.dumps( list(range(100000)), protocol=4 )      Enter      ←プロトコルレベル 4 を指定
>>> s5 = pickle.dumps( list(range(100000)), protocol=5 )      Enter      ←プロトコルレベル 5 を指定
```

このようにしてシリアルライズされたバイナリデータ s0~5 のデータサイズを調べる. (次の例)

例. データサイズを調べる (先の例の続き)

```
>>> len(s0) 788896
>>> len(s1) 368876
>>> len(s2) 368878
```

```
>>> len(s3) 368878
>>> len(s4) 368931
>>> len(s5) 368931
```

このように、プロトコルレベルに応じてシリアル化されたデータのサイズが異なることがあることがわかる。

dump の引数に 'protocol=' を与えてバイナリファイルを作成する例を pickle04-1.py に示す。これは 10,000 個の素数のリストを gmpy2 モジュール²¹⁵ を用いて作成し、それを各プロトコルレベルでシリアル化して別々のファイルとして保存するものである。

プログラム：pickle04-1.py

```
1 import pickle
2 import gmpy2
3
4 # 素数10,000個のリストを作成
5 n = 0
6 pLst = []
7 for _ in range(10000):
8     n = gmpy2.next_prime(n)
9     pLst.append( int(n) )
10
11 # プロトコルレベル毎にシリアル化して保存
12 for p in range(6):
13     fname = 'pickle04_'+str(p)+'.dat'
14     f = open(fname,'wb')
15     pickle.dump(pLst,f,protocol=p)
16     f.close()
```

このプログラムで作成したバイナリデータのファイルのプロトコルレベルとファイルサイズを調べるプログラムを pickle04-2.py に示す。

プログラム：pickle04-2.py

```
1 import pickletools
2 import os
3
4 # ファイルのプロトコルレベルを調べる関数
5 def pickleProtocol(path):
6     with open(path, 'rb') as f:
7         for op, arg, _ in pickletools.genops(f):
8             if op.name == 'PROTO':
9                 return arg
10    return None # プロトコル0/1 (PROTOがない場合) はNone
11
12 # ファイル毎に調査
13 print('file-name\tp-lev.\tfile-size\n-----')
14 for p in range(6):
15     fname = 'pickle04_'+str(p)+'.dat'
16     f = open(fname,'rb')
17     pl = pickleProtocol(fname)
18     f.close()
19     fs = os.path.getsize(fname)
20     print( f'{fname}\t{pl}\t{fs}' )
```

このプログラムを実行した例を次に示す。

実行例.

file-name	p-lev.	file-size

pickle04.0.dat	None	78988
pickle04.1.dat	None	36886
pickle04.2.dat	2	36888
pickle04.3.dat	3	36888
pickle04.4.dat	4	36896
pickle04.5.dat	5	36896

²¹⁵標準モジュールではないので、pip や conda などを用いてインストールする必要がある。(https://pypi.org/project/gmpy2/)

4.10.3 使用上の注意事項

先の pickle03class.py の例からわかるように、pickle でシリアライズされたバイナリデータを復元する際に、復元対象のオブジェクトのクラスのコンストラクタ（__init__）や __reduce__、__setstate__ といった特殊メソッドの中に記述されたコードが自動的に実行される。従って、クラス定義の中に悪意のあるコードが記述されていれば、データの復元時にそれらが実行されてしまう。従って、pickle の利用はセキュリティ上のリスクとなり得るので注意が必要である。

pickle は、Python 上で作成したオブジェクトをバイナリデータに変換するための簡便な方法であるが、素性のわからない第三者が pickle で作成したバイナリデータは、セキュリティ上の観点から復元するべきではない。従って pickle の利用は、Python 利用者が個人の利用の範囲に限る、あるいは、信頼できる緊密な開発者の間での利用に限るべきである。

上に述べたセキュリティ上のリスクは、ユーザが実装したクラス定義の内容から来るものであるが、標準ライブラリとして利用できる機能を応用して作られたバイナリデータは更に別の形でセキュリティリスクとなり得る。例えば次に示す pickleBgen.py で作成されるバイナリファイル pickleB.dat を復元する場合、クラス Bomb の定義ファイル pickleBgen.py が無くても、作成者が仕組んだコードが実行される。

プログラム：pickleBgen.py

```
1 import pickle, os
2
3 class Bomb:
4     def __reduce__(self):
5         # 復元時にこの関数と引数が呼ばれる
6         return ( os.system, ('echo [爆弾作動!]',) )
7
8 # シリアライズして保存
9 if __name__ == '__main__':
10     f = open('pickleB.dat', 'wb')
11     bad_pickle = pickle.dump(Bomb(), f)
12     f.close()
```

このプログラムを実行した結果として出来上がったバイナリファイル pickleB.dat を読み込んで復元する例を次に示す。

例. バイナリデータを復元するだけで実行されるコード

```
>>> import pickle  [Enter]    ←モジュールの読み込み
>>> f = open('pickleB.dat', 'rb')  [Enter]    ←ファイルを開いて
>>> m = pickle.load(f)  [Enter]    ←復元処理を行うだけで
[爆弾作動!]          ← os.system('echo [爆弾作動!]',) が実行される
>>> f.close()  [Enter]
```

os モジュールは Python の標準ライブラリであり、OS に関する様々な処理を実行する機能を提供する。上の実行例では、メッセージを表示するだけにとどまっているが、悪意を持ってこれを応用すると、システムに対する大きな脅威となる。

4.11 バイナリデータの作成と展開： struct モジュール

Python に標準的に添付されている struct モジュールを用いると、C 言語の**構造体** (struct) を作成、あるいは展開することができる。多くのアプリケーションソフトウェアは C 言語の構造体を標準的なデータ構造として採用しており、「バイナリデータ」と呼ばれるものは多くの場合において C 言語の構造体を意味する。従って、ファイル I/O や通信においてそのようなデータ構造を生成、展開するための機能が求められることがある。ここでは、struct モジュールの基本的な使用方法について説明する。

4.11.1 バイナリデータの作成

Python のバイト列としてバイナリデータを作成するには pack 関数を用いる。

書き方： struct.pack(フォーマット, データ列 ...)

連結対象の「データ列」は必要なだけコンマで区切って記述できる。第一引数の「フォーマット」には、変換後の C 言語のデータ型を意味する記述 (表 42) を与える。連結されて出来上がったバイト列が戻り値となる。

pack 関数の使用例を次に示す。

例. バイナリデータの作成

```
>>> import struct      Enter      ←モジュールの読み込み
>>> v1 = 1              Enter      ←ここから値 (3つ) を用意
>>> v2 = 2              Enter
>>> v3 = 3              Enter
>>> b = struct.pack( 'iii', v1, v2, v3 )  Enter  ←上記 3つの「整数」を連結
>>> b                  Enter      ←内容確認
b'\x01\x00\x00\x02\x00\x00\x03\x00\x00\x00'  ←結果表示 (バイト列)
```

変数 v1, v2, v3 の値を整数 'i' として連結してバイト列 b を得ている。フォーマットの記述も、連結対象の 3 個のデータに対応させて 'iii' としている。

表 42: C 言語の型を意味するフォーマットの表記 (一部)

表記	C 言語の型	意味	バイト長
c	char	長さ 1 のバイト値	1
b	signed char	長さ 1 のバイト値 (符号あり)	1
B	unsigned char	長さ 1 のバイト値 (符号なし)	1
?	Bool	真理値型	1
h	short	短い整数 (符号あり)	2
H	unsigned short	短い整数 (符号なし)	2
i	int	整数 (符号あり)	4
I	unsigned int	整数 (符号なし)	4
l	long	長い整数 (符号あり)	4
L	unsigned long	長い整数 (符号なし)	4
q	long long	更に長い整数 (符号あり)	8
Q	unsigned long long	更に長い整数 (符号なし)	8
e	(注 1)	浮動小数点数 (半精度)	2
f	float	浮動小数点数 (単精度)	4
d	double	浮動小数点数 (倍精度)	8
x	(注 2)	パディング	-

注 1: ニューラルネットの扱いにおいて多く用いられる型
注 2: C 言語の構造体としてメモリ境界を合わせるための調整領域

C 言語の char 型に関する注意：

Python の文字列型 (str 型) ではマルチバイト文字 (Unicode) も 1 つの文字の単位として扱えるが、C 言語の char 型は 1 バイトが 1 つの単位であるので注意すること。

先の例で作成したバイト列 `b` を展開する方法について説明する。

4.11.2 バイナリデータの展開

バイト列で表現されたバイナリデータを展開するには `unpack` 関数を用いる。

書き方： `struct.unpack(フォーマット, バイト列)`

展開されたデータのタプルが戻り値として得られる。 `unpack` 関数の使用例を次に示す。

例. バイナリデータの展開（先の例の続き）

```
>>> x = struct.unpack( 'iii', b )  [Enter]  ←バイナリデータを展開
>>> x  [Enter]  ←内容確認
(1, 2, 3)  ←展開結果がタプルとして得られる
```

【サンプルプログラム】

多数のデータを連結してバイナリデータとして保存し、それを読み込んで展開するプログラムの例を `struct01.py` に示す。

プログラム： `struct01.py`

```
1  # coding: utf-8
2  import struct
3
4  d = [x for x in range(20)]      # 20個の整数
5  fmt = 'i' * 20                 # 20個のフォーマット
6
7  b = struct.pack( fmt, *d )     # '*d'で引数の並びとしてデータの要素を並べる
8
9  f = open( 'struct01.bin', 'wb' ) # バイナリファイルを作成して
10 f.write( b )                   # 書き込む
11 f.close()
12
13 f = open( 'struct01.bin', 'rb' ) # 上で作成したファイルを開き
14 b2 = f.read()                  # 読み込む
15 f.close()
16
17 d2 = struct.unpack( fmt, b2 )  # データを展開
18
19 print( '元のデータ：' )
20 print( d, '\n' )
21 print( '連結->保存->読み込み->展開：' )
22 print( d2 )
```

これは20個の整数列のリストを生成し、それをバイナリデータとして連結、保存したものを読み込んで、展開する例である。 `pack` 関数呼び出し時の引数に `'*d'` という記述をすることで、リストの全要素を引数の並びとしている。このような書き方に関しては「2.8.1.4 関数呼び出し時の引数に `*` を記述する方法」(p.139) で解説している。

このプログラムの実行結果の例を次に示す。

元のデータ：

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

連結->保存->読み込み->展開：

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)

このサンプルのような手法で、多量のデータをバイナリデータとして保存する、あるいは読み込んで展開することができる。

4.11.3 バイトオーダーについて

2バイト以上の長さで表現される数値データに関しては、記憶資源上でのバイト毎の格納順番に注意しなければならない。このことをC言語の `int` 型（4バイト整数）を例に挙げて説明する。

16,909,060 という4バイトの整数について考える。この値は $1 \times 256^3 + 2 \times 256^2 + 3 \times 256^1 + 4 \times 256^0$ であり、記憶資源上には1, 2, 3, 4というバイト値の並びとして表現される。この値を `pack` 関数でバイト列（バイナリデータ）

に変換すると次のようになる。

例. $1 \times 256^3 + 2 \times 256^2 + 3 \times 256^1 + 4 \times 256^0$ をバイト列に変換 (Intel Core i7, Windows10 で実行)

```
>>> n = 1*(256**3) + 2*(256**2) + 3*(256**1) + 4*(256**0) [Enter] ←値の生成
>>> n [Enter] ←内容確認
16909060 ←結果表示
>>> import struct [Enter] ←モジュールの読み込み
>>> b = struct.pack( 'i', n ) [Enter] ←バイナリデータに変換
>>> b [Enter] ←内容確認
b'\x04\x03\x02\x01' ←結果表示 (バイト値の並びが逆順に見える)
```

この例は Intel の CPU である Core i7 の計算機 (OS は Windows10) で実行したものであるが、バイト列に変換すると、バイト値の並びが逆順になっているように見える。これは、Intel 製 CPU である Core i7 のバイトオーダーがリトルエンディアンであることに起因する。

計算機が値をバイト列として記憶資源に格納する際の順序をバイトオーダー (図 37)²¹⁶ という。バイトオーダーは計算機のアーキテクチャによって異なり、Intel の x86 系とその互換 CPU ではリトルエンディアンである。また、モトローラ社の MC68000 系列の CPU ではビッグエンディアンである。ただし、Java の JVM では独自のアーキテクチャを実現しており、実行環境の CPU の種類に依らず Java 上では常にビッグエンディアンである。

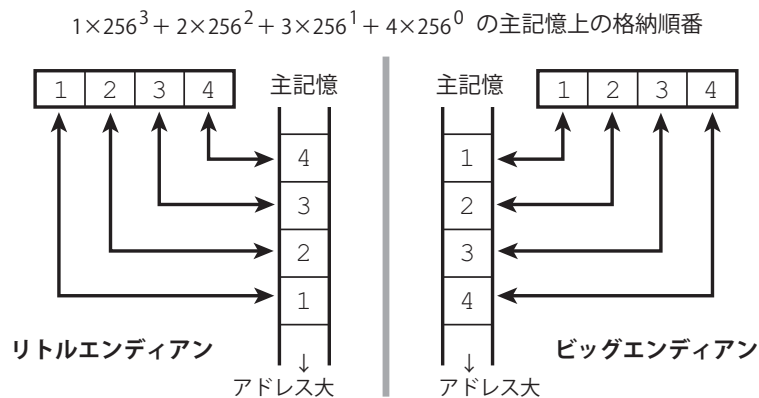


図 37: バイトオーダー

このような事情から、バイナリデータを異なるアーキテクチャの計算機環境の間で流用する際にはバイトオーダーに関して注意しなければならない。struct モジュールの pack / unpack 関数では、実行時にバイトオーダーを指定することが可能である。これに関して実行例を挙げて説明する。

例. バイトオーダーの指定 (先の例の続き)

```
>>> lb = struct.pack( '<i', n ) [Enter] ←リトルエンディアンを明に指定して変換
>>> lb [Enter] ←内容確認
b'\x04\x03\x02\x01' ←結果表示
>>> bb = struct.pack( '>i', n ) [Enter] ←ビッグエンディアンを明に指定して変換
>>> bb [Enter] ←内容確認
b'\x01\x02\x03\x04' ←結果表示
```

この例にあるように、型を指定するフォーマットに先立って '<' を付けるとリトルエンディアン、'>' を付けるとビッグエンディアンとして変換処理される。

▲注意▲

ARM, PowerPC の系列の CPU ではリトルエンディアン / ビッグエンディアンを切り替えることができる²¹⁷ ので、これら CPU の計算機環境との間でバイナリデータを交換する場合は特に意識すること。

²¹⁶リトルエンディアン、ビッグエンディアンの他にもミドルエンディアンなるバイトオーダーを採用している CPU も存在する。

²¹⁷バイエンディアン

4.12 バイナリデータをテキストに変換する方法：base64 モジュール

Base64 形式²¹⁸ は、バイナリデータを ASCII データしか通さない通信路で送受信する際に用いられるエンコーディングの形式である。Python には base64 モジュールが標準的に提供されており、このモジュールを利用することで、バイナリデータを Base64 形式のテキストデータに変換することができる。base64 モジュールは次のようにして Python 処理系に読み込む。

```
import base64
```

4.12.1 バイト列→Base64 データ

バイト列 (bytes 型) のデータを Base64 形式データに変換するには b64encode 関数を使用する。

例. バイト列 bd を Base64 データ b64 に変換する

```
b64 = base64.b64encode( bd )
```

得られた b64 の型も bytes である。

4.12.2 Base64 データ→バイト列

Base64 形式データをバイト列 (bytes 型) のデータに変換するには b64decode 関数を使用する。

例. Base64 データ b64 をバイト列 bd に変換する

```
bd = base64.b64decode( b64 )
```

4.12.3 サンプルプログラム

base64 モジュールの基本的な使用方法を、例を示しながら説明する。次に示すプログラム base64-01.py は、バイナリ形式のデータファイル python_icon.png (画像ファイル：図 38) を、Base64 形式のテキストファイル base64.dat に変換するものである。

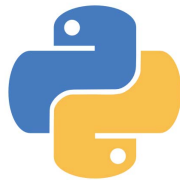


図 38: python_icon.png

プログラム：base64-01.py (バイナリファイルを Base64 ファイルに変換)

```
1  # coding: utf-8
2  import base64
3
4  # バイナリデータ (画像ファイル) の読み込み
5  f = open( 'python_icon.png', 'rb' )
6  bd = f.read()          # 一度に読み込み
7  f.close()
8
9  b64 = base64.b64encode( bd )    # Base64形式のバイト列に変換
10
11 # ファイルに保存
12 f = open( 'base64.dat', 'wb' )
13 f.write( b64 )
14 f.close()
```

base64-01.py によって作成された Base64 ファイル base64.dat の内容の一部を次に示す。

²¹⁸RFC4648

ファイル : base64.dat (内容の一部)

```
iVBORw0KGgoAAAANSUhEUgAAAGAAAAIACAYAAAD0eNT6AAAAAHNCVQICAgIfAhkiAAAAAwSFlzAAAN
1wAADdcBQiibeAAAAB1ORVhOU29mdHdhcmUAd3d3Lm1ua3NjYXB1Lm9yZ5vuPBoAACAAASURBVHic7d15
:
(途中省略)
:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAR/f/AYGSGo1CN4t4AAAAAE1FTkSuQmCC
```

このように、ASCII 文字で内容が構成されていることがわかる。

次に、これを復元してバイナリファイル base64.png を生成するプログラム base64-02.py を次に示す。

プログラム : base64-02.py (Base64 ファイルを復元)

```
1 # coding: utf-8
2 import base64
3
4 # Base64データの読み込み
5 f = open( 'base64.dat', 'rb' )
6 b64 = f.read()      # 一度に読み込み
7 f.close()
8
9 bd = base64.b64decode( b64 )    # Base64形式のバイト列を復元
10
11 # ファイルに保存
12 f = open( 'base64.png', 'wb' )
13 f.write( bd )
14 f.close()
```

このプログラムによって生成されたバイナリファイル base64.png の内容は図 38 に示したものと同一である。

base64 モジュールは、Base16、Base32、Base85 の各形式の変換機能も提供する。詳しくは公式ドキュメントを参照のこと。

4.13 編集可能なバイト列 : bytearray

先の「2.7.3.4 バイト列の扱い」(p.112) で解説したバイト列 (bytes 型) は **イミュータブル** なデータ構造であり、作成後はその内容を変更することができない。(次の例参照)

例. バイト列を変更しようとする試み

```
>>> b = b'abcdefg'  Enter    ← bytes 型のバイト列を作成
>>> b[1]  Enter          ← インデックス位置が 1 の要素を
98                        ← 参照はできるが...
>>> b[1] = 66  Enter      ← 値を更新しようとする...
Traceback (most recent call last):  ← エラー (例外) となる
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

これに対して、bytearray 型のバイト列は **ミュータブル** であり、変更が可能である。

4.13.1 bytearray の作成方法

bytearray を作成するにはいくつかの方法があるが、生成する長さ (バイト長) を指定する方法が最も単純である。

書き方 : bytearray(バイト長)

例. 5 バイトの長さの bytearray を作成

```
>>> b = bytearray( 5 )  Enter    ← bytearray を作成
>>> b  Enter            ← 内容確認
bytearray(b'\x00\x00\x00\x00\x00')  ← 作成された bytearray
```

この方法で作成された bytearray の初期値 (各バイト要素の値) は 0 である。また、bytearray オブジェクトは bytearray(バイト列表記) と表現される。

bytes 型の場合と同様の作成方法もある。

例. 文字列やバイト値のリストから作成

```
>>> b = bytearray( '日本語', 'utf-8' ) Enter ←文字列から bytearray を作成
>>> b Enter ←内容確認
bytearray(b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\xe9') ←内容
>>> b = bytearray( [65,66,67] ) Enter ←バイト値のリストから bytearray を作成
>>> b Enter ←内容確認
bytearray(b'ABC') ←内容
```

あるいはもっと単純に、bytes 型オブジェクトから bytearray を作成することもできる。

例. bytes 型オブジェクトを bytearray に変換する

```
>>> b = bytearray( b'abcdefg' ) Enter ← bytes オブジェクトをコンストラクタに与える
>>> b Enter ←内容確認
bytearray(b'abcdefg') ←内容
```

bytearray はミュータブルなので変更が可能である。

例. bytearray の内容変更（先の例の続き）

```
>>> b[1] Enter ←インデックス位置が 1 の要素の参照
98 ←参照可能
>>> b[1] = 66 Enter ←値の更新も可能
>>> b Enter ←内容確認
bytearray(b'aBcdefg') ←該当位置の要素が変更されている
```

4.13.2 他の型への変換

bytearray を bytes 型に変換するには 'bytes(bytearray オブジェクト)' とする。

例. bytes 型への変換（先の例の続き）

```
>>> bytes( b ) Enter ← bytes 型への変換
b'aBcdefg' ←変換結果
```

bytearray を文字列型に変換するには decode や str を使用する。

例. 文字列型への変換

```
>>> b = bytearray( '日本語', 'utf-8' ) Enter ← bytearray オブジェクトの作成
>>> b.decode('utf-8') Enter ←文字列型への変換 (1)
'日本語' ←変換結果
>>> str( b, 'utf-8' ) Enter ←文字列型への変換 (2)
'日本語' ←変換結果
```

4.13.3 ファイルへの出力

bytearray オブジェクトは bytes 型の場合と同様の方法でファイルに出力することができる。

例. bytearray オブジェクトをファイルに出力する（先の例の続き）

```
>>> f = open( 'binout01.dat', 'wb' ) Enter ←出力用ファイルをオープン
>>> f.write(b) Enter ← write メソッドによる出力
9 ←出力したバイト数
>>> f.close() Enter ←ファイルを閉じる
```

この処理によってファイル binout01.dat（下記参照）が作成される。

出力ファイル：binout01.dat

```
1 日本語
```

4.14 メモリ上でのファイル操作

io モジュールの BytesIO クラス、StringIO クラスは、メモリ上でファイルのようにデータを読み書きできる仕組み（仮想的なファイルの仕組み）を提供する。これにより、ディスク I/O に比べて高速にデータの一時保存や加工が行えるだけでなく、read や write などの標準的なファイル操作が利用できる。更に、一時的なデータの中継や、ファイルオブジェクトを引数に取る API との互換性が確保でき、プログラムの動作テストやデータ変換処理の効率化など、多くの利便性がもたらされる。また、実際にファイルを作成したり削除する必要がないため、リソース管理も容易である。BytesIO、StringIO は、実際のファイル I/O におけるバイナリモードとテキストモードにそれぞれ対応する。

通常のファイルは open する際、読み書きのモードを 'r'、'w' で指定するが、BytesIO、StringIO は基本的に**読み書き両用**である。また、BytesIO、StringIO はファイルシステム上のファイルではないので、それらインスタンスには使用できないメソッドも存在する。（例：fileno メソッド）

4.14.1 仮想的なテキストファイル：io.StringIO

StringIO クラスのインスタンスは通常のテキストモードのファイルオブジェクトと同様に扱うことができる。ただし、クラスは io.TextIOWrapper ではなく、あくまで StringIO である。このクラスのインスタンスの生成が、通常のテキストファイルに対する open の処理に相当する。また通常のファイルと同様に、StringIO インスタンスは close メソッドでその使用を終了することができる。

次の例は、StringIO に対して print 関数でテキストを出力するものである。

例. StringIO によるテキスト出力

```
>>> import io      Enter    ←モジュールの読み込み
>>> vf = io.StringIO() Enter    ←仮想テキストファイルの作成
>>> print('こんにちは。', file=vf) Enter    ←1行目のテキスト出力
>>> print('Python の StringIO を使っています。', file=vf) Enter    ←2行目のテキスト出力
```

上の例に続いて、アクセス位置をファイル先頭に移動してテキスト入力を行う例を示す。

例. StringIO によるテキスト入力（先の例の続き）

```
>>> vf.seek(0)      Enter    ←アクセス位置を先頭（0文字目）に移動
0                    ←移動した
>>> vf.readline()   Enter    ←テキストを1行読み込む
'こんにちは。 \n'    ←得られた内容
>>> vf.readline()   Enter    ←次のテキストを1行読み込む
'Python の StringIO を使っています。 \n' ←得られた内容
>>> vf.close()      Enter    ←仮想ファイルの使用終了
```

■ 初期値としての内容を与える方法

StringIO コンストラクタに初期値としての内容を与えることができる。

書き方： StringIO(文字列)

「文字列」を初期値として持つ仮想テキストファイルを生成して返す。

例. 初期値を持つ仮想テキストファイル

```
>>> import io      Enter    ←モジュールの読み込み
>>> vf = io.StringIO('一行目\n二行目\n') Enter    ←初期値を与えて仮想テキストファイルを作成
>>> vf.readline()   Enter    ←テキストを1行読み込む
'一行目\n'         ←得られた内容
>>> vf.readline()   Enter    ←次のテキストを1行読み込む
'二行目\n'         ←得られた内容
>>> vf.close()      Enter    ←仮想ファイルの使用終了
```

4.14.2 仮想的なバイナリファイル：io.BytesIO

BytesIO クラスのインスタンスは通常のバイナリモードのファイルオブジェクトと同様に扱うことができる。ただし、クラスは io.BufferedReader ではなく、あくまで BytesIO である。このクラスのインスタンスの生成が、通常の

バイナリファイルに対する open の処理に相当する。また通常のファイルと同様に、BytesIO インスタンスは close メソッドでその使用を終了することができる。

実際にバイナリデータ (bytes 型) を作成して BytesIO に対して出力した後、その内容を読み込む例を示す。

例. バイナリデータの作成

```
>>> bin = '日本語の文字列'.encode('utf-8') Enter ←マルチバイト文字を bytes 型 (バイナリ) に変換
>>> print(bin) Enter ←内容確認 ↓ bytes 型のデータが得られている
b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e\xe3\x81\xae\xe6\x96\x87\xe5\xad\x97\xe5\x88\x97'
```

これは、マルチバイト文字の列を encode メソッドで bytes 型に変換するもので、変数 bin にそれが得られている。次は、これを BytesIO に出力する例を示す。

例. BytesIO によるバイナリ出力 (先の例の続き)

```
>>> import io Enter ←モジュールの読み込み
>>> vf = io.BytesIO() Enter ←仮想バイナリファイルの作成
>>> vf.write(bin) Enter ←出力処理の実行
21 ←出力されたバイト数
```

これで、BytesIO にバイナリデータが出力された。次に、アクセス位置をファイル先頭に移動して、バイナリデータを読み込む例を示す。

例. BytesIO によるバイナリデータの入力 (先の例の続き)

```
>>> vf.seek(0) Enter ←アクセス位置を先頭 (0 バイト目) に移動
0 ←移動した
>>> bin2 = vf.read() Enter ←ファイルの内容を読み込む
>>> vf.close() Enter ←仮想ファイルの使用終了
>>> print(bin2) Enter ←内容確認 ↓ bytes 型のデータが得られている
b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e\xe3\x81\xae\xe6\x96\x87\xe5\xad\x97\xe5\x88\x97'
```

バイナリデータが得られていることがわかる。このデータを decode メソッドで文字列 (str) に復元する例を次に示す。

例. 得られたバイナリデータを文字列に復元する (先の例の続き)

```
>>> bin2.decode('utf-8') Enter ← bytes 型 (バイナリ) データをテキストに復元
'日本語の文字列' ←最初の文字列と同じものに復元できている
```

■ 初期値としての内容を与える方法

BytesIO コンストラクタに初期値としての内容を与えることができる。

書き方: BytesIO(バイナリデータ)

bytes 型の「バイナリデータ」を初期値として持つ仮想バイナリファイルを生成して返す。

例. 初期値を持つ仮想バイナリファイル

```
>>> import io Enter ←モジュールの読み込み
>>> vf = io.BytesIO(b'\xe6\xbc\xa2\xe5\xad\x97') Enter ←初期値を与えて仮想ファイルを作成
>>> vf.read().decode('utf-8') Enter ←バイナリデータの読み込みと文字列への変換
'漢字' ←読み取った内容を文字列に変換した結果
>>> vf.close() Enter ←仮想ファイルの使用終了
```

4.14.3 StringIO, BytesIO を使用する際の注意事項

StringIO, BytesIO はメモリ (実記憶) 上に仮想ファイルを作るので、アクセスは高速であるが、巨大なサイズのデータを扱うことには適していない。また StringIO, BytesIO はスレッドセーフではないので、複数スレッドで共有して使用する場合は、読み書きのタイミングに細心の注意を払ってプログラミングするか、あるいは排他制御を施す必要がある。(マルチプロセス環境下での共有においても同様)

4.15 exec と eval

exec 関数を用いると、文字列として記述された Python の文を実行することができる。

▲注意▲ 通信路やファイルから入力されたテキストデータを exec や後述の eval で直接実行するのはセキュリティ上のリスクとなるので避けること。

例. 文字列で表記された Python 文の実行

```
>>> s = 'print("これは文字列として記述した Python の文である。")' Enter ←文字列として文を記述
>>> exec(s) Enter ←文字列 s の内容を実行
これは文字列として記述した Python の文である。 ←実行結果
```

exec 関数で実行する文ではグローバル変数（大域変数）を使用することができる。

例. exec におけるグローバル変数の使用

```
>>> g = 1 Enter ←グローバル変数に値を設定
>>> s = 'print(g)' Enter ←文を記述
>>> exec(s) Enter ←文を実行
1 ←グローバル変数が引用されていることがわかる
>>> s = 'g = 5' Enter ←グローバル変数の値を変更する文を記述
>>> exec(s) Enter ←文を実行
>>> g Enter ←グローバル変数の内容確認
5 ←グローバル変数に変更されていることがわかる
```

exec 関数の戻り値は None である。

4.15.1 名前空間の指定

exec 関数によって Python の文を実行する際に、グローバル変数の名前空間を指定することができる。具体的には exec 関数の第 2 引数に辞書オブジェクトを与えることで、使用するグローバル変数に値を設定する。

例. 異なる名前空間としてグローバル変数を与える例（先の例の続き）

```
>>> exec('print(g)', {'g':3}) Enter ←グローバル変数 g に値 3 を設定して実行
3 ←結果表示
>>> g Enter ←元のグローバル変数 g の内容確認
5 ←元のグローバル変数は変更されていない
```

辞書オブジェクトに与える変数名は文字列型である。

exec の第 2 引数に与えているものは、厳密には 'globals=' という引数であり、先の例は次のようにしたものと同じである。

例. 上の例と同じ処理（先の例の続き）

```
>>> exec('print(g)', globals={'g':3}) Enter
3
>>> g Enter ←元のグローバル変数 g の内容確認
5
```

引数 'globals=' を省略すると、暗黙に globals=globals() が指定されたことになる。この globals() 関数²¹⁹ は真のグローバルの名前空間のオブジェクト群を返す。従って、このように引数を与えると、実際のグローバル変数にアクセスできる。（次の例）

例. 真のグローバル変数へのアクセス

```
>>> g = 5 Enter ←真のグローバル変数 g に値 5 を設定
>>> exec('g = 0; print(g)', globals=globals()) Enter ←真のグローバル変数群を使って実行
0 ←print 関数による出力
>>> g Enter ←真のグローバル変数 g の値を確認
0 ←exec の処理によって変更されている。
```

この例では、exec 関数に真のグローバル変数群を与えており、exec の実行において真のグローバル変数の g が書き

²¹⁹ 「4.20 使用されているシンボルの調査」(p.339) で解説する。

換えられていることが確認できる。

■ locals 引数による名前空間

exec には引数 'locals=' を与えることもでき、'globals=' とは別に、exec の実行時にのみ有効な、優先的に使用される変数を指定することができる。

例. locals 引数の指定

```
>>> g = 5; h = 7 [Enter]    ←グローバル変数 g, h に値を設定          ↓変数 h だけ一時使用
>>> exec('h = 1; print(f"{g},{h}")',globals=globals(),locals={'h':0}) [Enter]
g=5,h=1                ← h の値だけ locals のものが使われている
>>> print(f'{g},{h}') [Enter] ←終了後のグローバル変数の確認
g=5,h=7                ←元のまま
```

変数の参照は、まず locals 引数に与えられたものを探し、そこになければ globals 引数に与えられたものを探すという順序である。(locals → globals)

ただし、globals, locals 両方の引数を指定した場合、locals 引数に指定のない変数の値を exec 内で変更しようとするときのようなことが起こる。

例. 一見して理解しにくい事例

```
>>> g = 5; h = 7 [Enter]    ←グローバル変数 g, h に値を設定          ↓ g, h の両方の値を変更
>>> exec('g = 0; h = 1; print(f"{g},{h}")',globals=globals(),locals={'h':0}) [Enter]
g=0,h=1                ← exec 内で変更された通りの結果
>>> print(f'{g},{h}') [Enter] ←終了後のグローバル変数の確認
g=5,h=7                ←元のまま (exec 内では g も一時使用の変数として扱われた)
```

この例のように、グローバル変数 g の値を変更したように見えるが、複雑なことが起こるので注意が必要である。(この例では、g, h 両方とも locals の扱いとなっている)

locals 引数を与えずに、グローバル変数として未だ存在しない変数に exec 内で値を代入すると、それはグローバル変数となる。Python 処理系を新たに起動して次に示す例を実行する。

例. exec 内の処理によって新たに追加されるグローバル変数 (Python を再起動して実行)

```
>>> g = 5 [Enter]    ←グローバル変数 g のみに値を設定
>>> exec('g = 0; h = 1; print(f"{g},{h}")',globals=globals()) [Enter] ←新たな変数 h に値を代入
g=0,h=1                ← exec 内で変更された通りの結果
>>> print(f'{g},{h}') [Enter] ←終了後のグローバル変数の確認
g=0,h=1                ← h も新たなグローバル変数となっている
```

このように、引数 globals, locals を同時に指定する際には注意が必要である。

4.15.2 eval 関数

exec 関数と似た機能を持つ eval 関数がある。eval 関数で実行するものは「文」ではなく「式」であり、実行後はそれを評価した値を返す。名前空間、引数に関することは exec 関数に準じる。

例. eval 関数

```
>>> g = 5 [Enter]    ←グローバル変数に値を設定
>>> eval('2*g') [Enter] ←式を評価
10                ←戻り値
```

eval 関数でも使用するグローバル変数の名前空間を変更することができる。

例. 異なる名前空間としてグローバル変数を与える例 (先の例の続き)

```
>>> eval('2*g', {'g':15}) [Enter] ←グローバル変数 g に値 15 を設定して評価
30                ←評価結果
>>> g [Enter]    ←元のグローバル変数 g の内容確認
5                ←元のグローバル変数は変更されていない
```

4.16 collections モジュール

collections モジュールは様々なデータ構造を提供する。

4.16.1 キュー： deque

deque を使用すると、スタック（FILO）やキュー（FIFO）といったデータ構造を簡単に実現できる。また、リストを使用してこれらデータ構造を実現する場合と比べても処理速度が早い。deque の使用に先立って、次のようにして必要な機能を読み込む。

```
from collections import deque
```

deque オブジェクトを作成するには、コンストラクタ deque() を呼び出す。コンストラクタの引数にリストなどのデータ列を与えることで、deque オブジェクトに初期値を与えることができる。引数を省略すると空の deque オブジェクトが作成される。

4.16.1.1 要素の追加と取り出し： append, pop

deque オブジェクトに対して append メソッドを実行することで、その右端に要素を追加することができる。また、pop メソッドを実行すると、deque オブジェクトの右端から要素を取り出して返す。その後、その要素は deque オブジェクトから削除される。

例. deque によるスタックの実現

```
>>> from collections import deque  Enter    ←モジュールの読み込み
>>> q = deque()  Enter    ← 空の deque オブジェクトの作成
>>> q.append('a')  Enter    ← 右端に要素を追加
>>> q  Enter    ← 内容確認
deque(['a'])    ← deque オブジェクトの状態
>>> q.append('b')  Enter    ← 右端に要素を追加
>>> q  Enter    ← 内容確認
deque(['a', 'b'])    ← deque オブジェクトの状態
>>> q.append('c')  Enter    ← 右端に要素を追加
>>> q  Enter    ← 内容確認
deque(['a', 'b', 'c'])    ← deque オブジェクトの状態
>>> q.pop()  Enter    ← 右端の要素を取り出して削除
'c'    ← 戻り値
>>> q  Enter    ← 内容確認
deque(['a', 'b'])    ← deque オブジェクトの状態
>>> q.pop()  Enter    ← 右端の要素を取り出して削除
'b'    ← 戻り値
>>> q  Enter    ← 内容確認
deque(['a'])    ← deque オブジェクトの状態
>>> q.pop()  Enter    ← 右端の要素を取り出して削除
'a'    ← 戻り値
>>> q  Enter    ← 内容確認
deque([])    ← deque オブジェクトの状態
```

append, pop と似たメソッドに appendleft, popleft というメソッドもあり、それらは deque の左端に対して要素の追加や取り出しの処理を行う。

課題. ここで説明したメソッド群を用いてキュー（FIFO）を実現せよ。

4.16.1.2 要素の順序の回転： rotate

deque オブジェクトの要素を失うことなくその順序をずらす（回転する）には rotate メソッドを用いる。このメソッドの引数には回転のステップ数を与える。（ステップ数の暗黙値は 1 である） 正のステップ数を与えると右に、負のス

テップ数を与えると左に回転する。

例. deque オブジェクトの回転

```
>>> q = deque( ['a','b','c','d','e'] )  Enter    ← deque オブジェクトの作成
>>> q  Enter    ← 内容確認
deque(['a', 'b', 'c', 'd', 'e'])    ← deque オブジェクトの状態
>>> q.rotate()  Enter    ← 右に 1 ステップ回転
>>> q  Enter    ← 内容確認
deque(['e', 'a', 'b', 'c', 'd'])    ← deque オブジェクトの状態
>>> q.rotate(2)  Enter    ← 右に 2 ステップ回転
>>> q  Enter    ← 内容確認
deque(['c', 'd', 'e', 'a', 'b'])    ← deque オブジェクトの状態
>>> q.rotate(-3)  Enter    ← 左に 3 ステップ回転
>>> q  Enter    ← 内容確認
deque(['a', 'b', 'c', 'd', 'e'])    ← deque オブジェクトの状態
```

4.16.2 要素の集計：Counter

Counter を使用すると、データ構造の要素の数を集計することができる。Counter の使用に先立って、次のようにして必要な機能を読み込む。

```
from collections import Counter
```

例. 要素数の集計

```
>>> from collections import Counter  Enter    ←モジュールの読み込み
>>> c = Counter( ['a','b','a','c','a','b'] )  Enter    ←リストの要素数を集計
>>> c  Enter    ←内容確認
Counter({'a': 3, 'b': 2, 'c': 1})    ←集計結果
```

集計結果は Counter オブジェクトとして得られる。これは辞書 (dict) のサブクラスであり、辞書に対するメソッドが使用できる。

例. 戻り値は辞書と同様の扱いが可能 (先の例の続き)

```
>>> c['a']  Enter    ←要素 'a' の個数を求める
3          ← 3 個
>>> list( c.keys() )  Enter    ←キーのリストを求める
['a', 'b', 'c']      ←キーのリスト
>>> list( c.values() )  Enter    ←値のリストを求める
[3, 2, 1]            ←値のリスト
>>> list( c.items() )  Enter    ←全エントリのリストを求める
[('a', 3), ('b', 2), ('c', 1)] ←全エントリのリスト
```

同様の方法で、タプルや文字列の要素も集計できる。(次の例参照)

例. 要素数の集計 (先の例の続き)

```
>>> Counter( ('a','b','a','c','a','b') )  Enter    ←タプルの要素数を集計
Counter({'a': 3, 'b': 2, 'c': 1})    ←集計結果
>>> Counter( 'abacab' )  Enter    ←文字列の要素数を集計
Counter({'a': 3, 'b': 2, 'c': 1})    ←集計結果
```

4.16.2.1 出現頻度の順に集計結果を取り出す

集計結果の Counter オブジェクトに対して most_common メソッドを使用すると、出現頻度の高いものを取り出すことができる。

例. 出現頻度の高いものを取り出す (先の例の続き)

```
>>> c.most_common(1)  [Enter]    ←最も出現頻度の高いものを抽出
[('a', 3)]             ←'a' の出現頻度が最も高い
>>> c.most_common(2)  [Enter]    ←出現頻度の高いものを2位まで抽出
[('a', 3), ('b', 2)]   ←'a' が1位, 'b' が2位
>>> dict( c.most_common(2) ) [Enter] ←上の結果を辞書に変換
{'a': 3, 'b': 2}       ←これで辞書として使える
```

most_common の引数を空にすると、全ての集計結果を頻度の高い順に並べたものを返す。

例. 集計結果を出現頻度の高い順に全て取り出す (先の例の続き)

```
>>> c.most_common()  [Enter]    ←引数を空にして実行
[('a', 3), ('b', 2), ('c', 1)]  ←結果
```

4.16.3 namedtuple

namedtuple を使用すると、class 文によるクラス定義に依ることなく、**ドット表記の属性**²²⁰ の保持を実現することができる。ただし、namedtuple のオブジェクトはタプルと同様、事後に値を変更することはできない。

namedtuple の使用に先立って、次のようにして必要な機能を読み込む。

```
from collections import namedtuple
```

例として「3つの属性 x,y,z を持つオブジェクト型」である 'MyNt' 型を定義して使用する流れを示す。

例. 3つの属性 x,y,z を持つ 'MyNt' 型の定義

```
>>> from collections import namedtuple [Enter]    ←モジュールの読み込み
>>> MyNt = namedtuple('MyNt', ['x', 'y', 'z']) [Enter]    ←'MyNt' 型の定義
```

このように、

```
namedtuple(型名, 属性リスト)
```

とすることで「型」が生成される。「型名」と「属性リスト」の要素は文字列型で与える。この例では、'MyNt' という型を生成して、それを MyNt というシンボルに割り当てている。これ以後は MyNt を型としてオブジェクトを生成することができる。(次の例参照)

例. MyNt 型オブジェクトの生成 (先の例の続き)

```
>>> nt = MyNt( x=1, y=2, z=3 ) [Enter]    ← MyNt 型オブジェクト nt の生成
>>> nt.x [Enter]    ← nt の x プロパティを参照
1      ←値が確認できる。
```

このように、

型名 (値の設定)

と記述することで、属性に値を与えてオブジェクトを生成することができる。

例. オブジェクトの構造の確認 (先の例の続き)

```
>>> nt [Enter]    ←オブジェクト自体 (全体) を確認する
MyNt(x=1, y=2, z=3)    ←データ構造が確認できる。
>>> type( nt ) [Enter]    ← nt の型を確認する
<class '__main__.MyNt'>    ←結果表示
```

オブジェクトを生成する際は、全ての属性に初期値を与えなければならない。

例. オブジェクト生成時に属性値の設定が不足してはならない (先の例の続き)

```
>>> MyNt( x=1, y=2 ) [Enter]    ←初期値不足での MyNt 型オブジェクト生成の試み
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
TypeError: MyNt.__new__() missing 1 required positional argument: 'z'
```

²²⁰ 「属性」は「プロパティ」とも呼ばれる。

4.17 itertools モジュール

itertools モジュールは反復処理のためのイテレータを作成する便利な機能を多数提供する。ここではその中からいくつかの機能を紹介し、それらの基本的な使用方法について解説する。

4.17.1 イテラブルの連結：chain

chain クラスを使用すると、複数のイテラブルを形式的に連結することができる。

書き方： chain(イテラブルの並び)

「イテラブルの並び」は1つ以上のオブジェクトをコンマで区切って並べたもので、それらを形式的に連結したイテレータ (chain オブジェクト) を返す。引数に与えるイテラブルは、リストや文字列といったイテラブルオブジェクトの他に、ファイルオブジェクトなど様々なものを使用できる。

例. イテラブルの連結

```
>>> from itertools import chain  [Enter]    ← chain クラスの読み込み
>>> q1 = ['a','b']; q2 = 'cd'  [Enter]    ←イテラブルオブジェクトを2つ用意
>>> for x in chain(q1,q2):  [Enter]    ←それらを連結して反復処理に使用する
...     print(x)  [Enter]
...  [Enter]
a      ←出力 (ここから)
b
c
d      ←出力 (ここまで)
```

別々に用意されたイテラブルオブジェクト q1, q2 が連結され1つのイテレータとして扱われていることがわかる。

chain オブジェクトはイテラブルであるだけでなく イテレータである。(次の例)

例. イテレータとしての chain オブジェクト (先の例の続き)

```
>>> c = chain(q1,q2)  [Enter]    ← chain オブジェクト c の作成
>>> c  [Enter]    ←内容確認
<itertools.chain object at 0x000001DB14EBE020>    ← chain オブジェクトになっている
>>> next(c)  [Enter]    ← next 関数の実行
'a'      ←要素が得られている
```

next 関数で要素を取り出すことができおり、chain オブジェクトがイテレータであることがわかる。

4.17.2 無限のカウンタ：count

count クラスを用いると無限長のイテレータが実現できる。

書き方： count(start=初期値, step=増分)

「初期値」から「増分」ずつ増やしながら次々と値を生成する count オブジェクト (イテレータの一種) を返す。count(初期値, 増分) と記述しても良い。

例. 無限のカウンタ

```
>>> from itertools import count  [Enter]    ← count クラスの読み込み
>>> for x in count(0,2):  [Enter]    ← 0 から 2 間隔で値を際限なく生成
...     s = input(str(x)+' : q で終了> ')  [Enter]    ← 'q' が入力されるまで
...     if s == 'q': break  [Enter]    ← 際限なく繰り返す
...  [Enter]
0 :  q で終了> a  [Enter]    ←入力インタラクション (ここから)
2 :  q で終了> b  [Enter]
4 :  q で終了> q  [Enter]    ←入力インタラクション (ここまで)
```

この例は、キーボードからの入力受けを繰り返すもので、'q' が入力されると for の繰り返しを終了する。count オブジェクトが生成した数値が input 関数のプロンプトに表示されていることが確認できる。

4.17.3 イテラブルの繰り返し： cycle

cycle クラスを使用すると、与えられたイテラブルを無限に繰り返すイテレータ（cycle オブジェクト）を生成することができる。

書き方： cycle(イテレータ)

「イテレータ」を無限に繰り返すイテレータを返す。

例. イテレータを際限なく繰り返す

```
>>> from itertools import cycle  [Enter]    ← cycle クラスの読み込み
>>> q = ['a','b','c']  [Enter]    ←このようなイテレータ q を
>>> for n,x in enumerate( cycle(q) ):  [Enter]    ←際限なく繰り返す
...     if n < 20:  [Enter]    ←ただし便宜上、20 回で終了させる
...         print(x,end=' ')  [Enter]
...     else:  [Enter]
...         print()  [Enter]
...         break  [Enter]
...  [Enter]
a b c a b c a b c a b c a b c a b c a b    ←出力
```

▲注意▲

cycle オブジェクトは与えられたイテレータの要素を内部に記録することで要素の生成を繰り返す。従って、長大なイテレータをコンストラクタに与える場合は注意が必要である。従って cycle クラス以外のものを状況に応じてプログラマが独自に実装する方が好ましい場合もある。

4.17.4 オブジェクトの繰り返し： repeat

repeat クラスを使用すると、与えられたオブジェクトを指定した回数だけ繰り返すイテレータ（repeat オブジェクト）を生成することができる。

書き方： repeat(オブジェクト, times=回数)

「オブジェクト」を「回数」だけ繰り返すイテレータを返す。「回数」を省略すると無限に繰り返すイテレータを返す。repeat(オブジェクト, 回数) と記述しても良い。

例. 'a' を 10 回繰り返すイテレータ

```
>>> from itertools import repeat  [Enter]    ← repeat クラスの読み込み
>>> rep = 10  [Enter]    ←繰り返し回数
>>> for n,x in enumerate( repeat('a',rep) ):  [Enter]
...     if n < rep-1:  [Enter]
...         print(x,end=' ')  [Enter]
...     else:  [Enter]
...         print(x)  [Enter]
...  [Enter]
a a a a a a a a a a    ←出力
```

4.17.5 連続要素のグループ化： groupby

groupby クラスを使用すると、与えられたイテラブルを連続要素ごとにグループ化し、それらが並ぶイテレータ（groupby オブジェクト）を生成することができる。

書き方： groupby(イテラブル)

「イテラブル」を連続要素ごとにグループ化し、(要素, 部分列イテレータ) なる形式の要素が並ぶイテレータにして返す。この場合の「部分列イテレータ」は `_grouper` オブジェクトである。

例. 連続要素のグループ化

```
>>> from itertools import groupby  [Enter]    ← groupby クラスの読み込み
>>> q = ['a','a','a','a','b','b','b','c','c','d']  [Enter]    ←連続要素を持つイテラブル
>>> for e,x in groupby(q):  [Enter]    ←グループごとに取り出しながら
...     print(e,': ',end='')  [Enter]
...     for y in x:  [Enter]    ←部分列イテレータごとに
...         print(y,end=' ')  [Enter]    ←出力する
...     print()  [Enter]
...  [Enter]
a :  a a a a    ←出力（ここから）
b :  b b b
c :  c c
d :  d          ←出力（ここまで）
```

この例で作成したイテラブル q がどのようにグループ化されたかを更に次のような形で確認するとわかりやすい。

例. 連続要素のグループ化：その2（先の例の続き）

```
>>> [ (e,[y for y in x]) for e,x in groupby(q) ]  [Enter]    ←リストの内包表記で確認する
[('a', ['a', 'a', 'a', 'a']), ('b', ['b', 'b', 'b']), ('c', ['c', 'c']), ('d', ['d'])]
```

groupby クラスは指定した「ある特徴」で連続要素をグループ化することもできる。

書き方： groupby(イテラブル, key=特徴化関数)

要素の特徴を取得する関数を「特徴化関数」に与える。

次の例に示す関数 ft は、与えられた 0 から 29 までの数の所属範囲を '0～9', '10～19', '20～29' という文字列で特徴化するものである。

例. 与えられた数の所属範囲を返す関数（先の例の続き）

```
>>> def ft(x):  [Enter]
...     if x<10: return '0～9'  [Enter]
...     if x<20: return '10～19'  [Enter]
...     return '20～29'  [Enter]
...  [Enter]
>>>  ← Python のプロンプトに戻った
```

この関数 ft を groupby コンストラクタの引数「key=」に与えてグループ化する例を次に示す。

例. 特徴ごとに要素をグループ化する（先の例の続き）

```
>>> q = range(30)[::-1]  [Enter]    ← 29～0 の数列を作成
>>> for e,x in groupby(q,key=ft):  [Enter]    ←関数 ft で特徴化して反復処理
...     print(e,': ',end='')  [Enter]
...     for y in x:  [Enter]
...         print(y,end=' ')  [Enter]
...     print()  [Enter]
...  [Enter]
20～29 :  29 28 27 26 25 24 23 22 21 20    ←出力（ここから）
10～19 :  19 18 17 16 15 14 13 12 11 10
0～9   :   9 8 7 6 5 4 3 2 1 0            ←出力（ここまで）
```

4.17.6 直積集合：product

product クラスを使用すると、与えられた複数のイテラブルの直積集合（**デカルト積**）をイテレータとして生成することができる。

書き方： product(イテラブルの並び)

「イテラブルの並び」の直積集合を作り、イテレータ（product オブジェクト）として返す。

例. 直積集合

```
>>> from itertools import product  ← product クラスの読み込み
>>> q1 = [1,2,3]; q2 = ['a','b','c']  ← 2つのイテラブルを用意
>>> for x in product(q1,q2):  ← 直積集合を作って
...     print(x,end=' ')  ← 出力する
... else: print() 
...  ↓ 出力
(1, 'a') (1, 'b') (1, 'c') (2, 'a') (2, 'b') (2, 'c') (3, 'a') (3, 'b') (3, 'c')
```

この例のように、直積集合の要素がタプルの形で得られている。

課題. 3つ以上のイテラブルの直積集合を作成し、上記の例に倣って出力を確認せよ。

4.17.7 組合せ: combinations

combinations クラスを使用すると、与えられたイテラブルの要素の組合せをイテレータとして生成することができる。

書き方: combinations(イテラブル, r=個数)

「イテラブル」の要素から「個数」だけ要素を取り出す組合せを作り、それをイテレータ (combinations オブジェクト) として返す。

combinations(イテラブル, 個数) と記述しても良い。

例. 組合せ

```
>>> from itertools import combinations  ← combinations クラスの読み込み
>>> q = ['a','b','c','d']  ← このイテラブルから
>>> for x in combinations(q,2):  ← 2つの要素を取り出す組合せを作り
...     print(x,end=' ')  ← 出力する
... else: print() 
...  ↓ 出力
('a', 'b') ('a', 'c') ('a', 'd') ('b', 'c') ('b', 'd') ('c', 'd')
```

4.17.8 順列: permutations

permutations クラスを使用すると、与えられたイテラブルの要素の順列をイテレータとして生成することができる。

書き方: permutations(イテラブル, r=個数)

「イテラブル」の要素から「個数」だけ要素を取り出す順列を作り、それをイテレータ (permutations オブジェクト) として返す。

permutations(イテラブル, 個数) と記述しても良い。

例. 順列

```
>>> from itertools import permutations  ← permutations クラスの読み込み
>>> q = ['a','b','c','d']  ← このイテラブルから
>>> for x in permutations(q,2):  ← 2つの要素を取り出す順列を作り
...     print(x,end=' ')  ← 出力する
... else: print() 
...  ↓ 出力
('a', 'b') ('a', 'c') ('a', 'd') ('b', 'a') ('b', 'c') ('b', 'd')
('c', 'a') ('c', 'b') ('c', 'd') ('d', 'a') ('d', 'b') ('d', 'c')
```

4.18 列挙型： enum モジュール

列挙型のデータ構造を提供する enum モジュールが Python 3.4 から標準的に提供されている。列挙型は、順序のある要素から成るデータ列であり、for 文などのイテレーション（繰り返し制御）に使用することができる²²¹。列挙型の要素は名前と値を持つ。

4.18.1 Enum 型

enum モジュールが提供する最も基本的なデータ型に Enum クラスがある。Enum を使用するには

```
from enum import Enum
```

として必要なものを読み込む。その後、次の様な記述で Enum オブジェクトを生成する。

書き方： Enum('型名', 要素の列)

「要素の列」には登録する各要素の名前のリスト（あるいは空白で区切られた文字列）を与える。各要素には 1 から始まる整数値（1, 2, 3, …）が暗黙値として設定される。

'a', 'b', 'c' という名前を持つ 3 つの要素から成る Enum オブジェクト E を生成する例を示す。

例. 列挙型オブジェクト E の生成

```
>>> from enum import Enum      Enter    ←モジュールの読み込み
>>> E = Enum('MyEnum', ['a', 'b', 'c']) Enter    ←'MyEnum' という型名の Enum オブジェクトを生成
>>> E      Enter    ←内容確認
<enum 'MyEnum'>      ←列挙型となっていることがわかる
```

この例と同じ Enum オブジェクトは

```
E = Enum('MyEnum', 'a b c')
```

として生成することもできる。

Enum オブジェクトの要素にアクセスするには、当該オブジェクトの後ろにドット「.」と要素の名前を付けるか、「[名前]」を添える。また、要素の名前と値は name, value という属性（プロパティ）として参照できる。

例. Enum の要素へのアクセス（先の例の続き）

```
>>> E.a      Enter    ←名前 'a' の要素の内容を確認
<MyEnum.a: 1>      ←結果表示（名前が'a', 値が 1 となっている）
>>> E.a.name  Enter    ←明に名前を取り出す
'a'      ←結果表示
>>> E.a.value  Enter    ←要素の値を取り出す
1      ←結果表示
>>> E['a'].value  Enter    ←'[名前]' による要素の参照
1      ←同じ結果
```

Enum オブジェクトの要素の型は <enum ユーザが定義した型> である。

例. Enum 要素の型（先の例の続き）

```
>>> type( E.a )  Enter    ←型の調査
<enum 'MyEnum'>      ←このような型
```

Enum オブジェクトに存在しない名前を用いてアクセスするとエラーが発生する。

例. 存在しない名前でアクセス（先の例の続き）

```
>>> E.d      Enter    ←名前 'd' の要素にアクセスを試みると…
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    (途中省略)
    raise AttributeError(name) from None
AttributeError: d
```

²²¹列挙型はイテレータ（p.92 「2.6.1.8 イテレータ」参照のこと）ではないので参照によって破壊されない。

この例では `AttributeError` が起こっている。同様に「[名前]」を添える形で存在しない名前の要素にアクセスしようとすると `KeyError` が起こる。

イテレーション（for 文などによる繰り返し処理）において Enum の全要素に順番にアクセスすることができる。

例. for 文で Enum オブジェクトの要素に順次アクセスする（先の例の続き）

```
>>> for e in E: Enter      ← for 文で各要素に順番にアクセスする
...     print( e.name, '->', e.value ) Enter      ←各要素の名前と値を表示
... Enter      ← for 文の終了
a -> 1      ←名前と値を順番に表示
b -> 2
c -> 3
```

Enum オブジェクトの各要素の名前と値を辞書の形で与えることができる。

例. 名前と値のペアを辞書の形で与える

```
>>> E2 = Enum('MyEnum2', {'x':100, 'y':200, 'z':300}) Enter      ←辞書で名前と値を与える
>>> E2.y.value Enter      ←値の確認
200      ←結果表示
```

4.18.1.1 class 定義による Enum オブジェクトの作成

列挙型オブジェクト（Enum オブジェクト）を作成する場合、先に示した方法よりも class 宣言によるクラス定義による方法が推奨される。

書き方： class 列挙型オブジェクト (Enum):
要素 1 = 値 1
要素 2 = 値 2
:

先に例示した E2 は次のようにして作成することもできる。

例. class 宣言による列挙型オブジェクトの作成（先の例の続き）

```
>>> class E2(Enum): Enter      ←定義の開始
...     x = 100 Enter
...     y = 200 Enter
...     z = 300 Enter
... Enter      ←定義の記述の終了
>>> E2.x.value Enter      ←値の確認
100      ←結果表示
```

4.18.2 定数の取り扱いを実現する方法の例

多くのプログラミング言語では定数を定義して使用することができる。この場合の定数は値の変更ができないものであり、値を持った変数とは異なるものである。

Python の基本的な文法には定数を実現するためのものはないが、enum モジュールを用いることで定数を実現することができる。ただし、Enum クラスによって実現した定数の扱いには注意する必要がある。例えば先の例で作成した E2 というクラスは x, y, z という 3 つの定数を持ち、それらの値は整数型 (int) であるが、それらを用いて整数型の演算をしようすると問題が起こる。(次の例)

例. Enum の値の取り扱い上の問題（先の例の続き）

```
>>> E2.y + E2.z Enter      ← Enum の値を演算に用いる試み
Traceback (most recent call last):      ←エラーが発生する
  File "<stdin>", line 1, in <module>
    E2.y + E2.z
    ~~~~~
TypeError: unsupported operand type(s) for +: 'E2' and 'E2'
```

これは、先に定義した定数クラス MyEnum2 に '+' の演算が定義されていないことが原因である。従って、整数型と同じ処理を Enum の定数にも適用するには、整数型 (int) と Enum の両方を継承するクラスを定義して、それを

整数型の定数のクラスとして用いると良い。(次の例)

例. 整数の定数を定義する

```
>>> class SI(int,Enum): Enter ← int と Enum の両方を継承するクラスの定義
...     KiB = 1024 Enter
...     MiB = 1048576 Enter
... Enter ←クラス定義の終了
```

このように SI クラスを定義すると、int 型と Enum クラス両方の性質を継承した要素が扱える。例の中では KiB, MiB という2つの整数型の要素がクラスのプロパティとして定義されているが、それらは同時に Enum の要素にもなる。これら要素を整数 (int 型) の定数と見なして扱うことができる。

例. 計算や比較への応用 (先の例の続き)

```
>>> SI.KiB * SI.KiB Enter ←要素同士の計算が可能
1048576 ←結果表示
>>> SI.KiB * SI.KiB == SI.MiB Enter ←比較も可能
True ←結果表示
```

Enum の要素の値は事後に変更することができない。

例. 値の変更の試み (先の例の続き)

```
>>> SI.KiB = 1000 Enter ←要素の値を変更しようとする...
Traceback (most recent call last): ←エラーが発生する
  File "<stdin>", line 1, in <module>
    (途中省略)
    raise AttributeError('cannot reassign member %r' % (name, ))
AttributeError: cannot reassign member 'KiB'
```

例. イテレーション (繰り返し処理) の対象としても利用できる (先の例の続き)

```
>>> for e in SI: Enter ← for 文で各要素に順番にアクセスする
...     print( e.name, '->', e.value ) Enter ←各要素の名前と値を表示
... Enter ← for 文の終了
KiB -> 1024 ←名前と値を順番に表示
MiB -> 1048576
```

上の例で示した SI のようなクラスは IntEnum クラスとして用意されている。

4.18.3 IntEnum

整数型 (int 型) と Enum の両方を継承した IntEnum クラスが使用できる。このクラスの使用に先立って 'from enum import IntEnum' として必要なものを読み込む。

例. IntEnum クラス

```
>>> from enum import IntEnum Enter ←モジュールの読み込み
>>> class SI2(IntEnum): Enter ← IntEnum を継承する SI2 クラスの定義
...     KiB = 1024 Enter
...     MiB = 1048576 Enter
... Enter ←クラス定義の記述の終了
>>> SI2.KiB * SI2.KiB Enter ←整数としての演算
1048576 ←結果表示
```

これと同様の処理を次のようにしても実行できる。

例. IntEnum クラスのインスタンスを生成

```
>>> SI2 = IntEnum('SI', {'KiB':1024, 'MiB':1048576}) Enter ← IntEnum のインスタンス
>>> SI2.KiB * SI2.KiB Enter ←整数としての演算
1048576 ←結果表示
```

4.18.4 auto 関数による Enum 要素への値の割り当て

Enum を継承したクラスを定義する際、各要素の値に auto 関数で自動的に値を与えることができる。auto 関数を使用するには

```
from enum import auto
```

として Enum モジュールから読み込んでおく必要がある。

例. auto 関数による値の割り当て

```
>>> from enum import Enum, auto  Enter    ←モジュールの読み込み
>>> class MyInt(int,Enum):  Enter    ←クラスの定義
...     a = 10  Enter    ←値として 10 を割り当てる
...     b = auto()  Enter    ← auto による値の割り当て
...     c = auto()  Enter    ← auto による値の割り当て
...  Enter    ←クラスの定義の記述の終了
```

この例で定義している MyInt は3つの要素 a, b, c を持ち、a には 10 が割り当てられ、後の要素には auto 関数で値が割り当てられている。auto 関数は 1 刻みで増加する形で値を順次生成する。

例. 各要素の値を調べる（先の例の続き）

```
>>> for e in MyInt:  Enter    ← for 文による繰り返しで全要素を調べる
...     print( e.name, '->', e.value )  Enter    ←各要素の名前と値を表示
...  Enter    ← for 文の記述の終了

a -> 10    ← 10 から 1 ずつ増える形で値が設定されている。
b -> 11
c -> 12
```

先頭の要素も auto 関数で値を設定した場合は 1 から始まる整数が順番に割り当てられる。

4.18.4.1 非数値要素への auto 関数による値の割り当て（参考事項）

文字列（str 型）と Enum 型を継承する次のようなクラスを定義する場合について考える。

例. str と Enum を継承するクラス

```
>>> class MyStr(str,Enum):  Enter    ←クラスの定義
...     a = 'start'  Enter    ←値として 'start' を割り当てる
...     b = auto()  Enter    ← auto による値の割り当て
...     c = auto()  Enter    ← auto による値の割り当て
...  Enter    ←クラスの定義の記述の終了
```

Python 3.12 までは MyStr をこのように定義することもできるが推奨されず、次のような警告メッセージが出力される。

警告メッセージ：Python 3.12 での実行

```
<stdin>:3: DeprecationWarning: In 3.13 the default 'auto()'/ '_generate_next_value_'
will require all values to be sortable and support adding +1
and the value returned will be the largest value in the enum incremented by 1
<stdin>:4: DeprecationWarning: In 3.13 the default 'auto()'/ '_generate_next_value_'
will require all values to be sortable and support adding +1
and the value returned will be the largest value in the enum incremented by 1
```

この例のような定義は Python 3.13 以降の版では許されない（エラーとなる）ので注意すること。

Python 3.12 以前に限定：上で定義した MyStr の各要素の値を調べる。

例. 各要素の値を調べる（先の例の続き）

```
>>> for e in MyStr:  Enter    ← for 文による繰り返しで全要素を調べる
...     print( e.name, '->', e.value, 'type:', type(e.value) )  Enter    ←各要素の名前と
...  Enter    ← for 文の記述の終了                                値と型を表示

a -> start type: <class 'str'>    ← 最初は 'start'
b -> 1 type: <class 'str'>        ← 次の値（文字列としての '1'）
c -> 2 type: <class 'str'>        ← 更に次の値（文字列としての '2'）
```


auto 関数による値の生成は '1' から始まっていることがわかる。

4.18.5 ビットフラグ：Flag

enum.Flag を用いるとビットフィールドの取り扱いが実現できる。これについて、例を挙げて解説する。

次に示す例は、Unix系 OS におけるファイルパーミッション「`rwX`」(Read/Write/eXecute) のフラグを `FilePermission` というクラスとして定義するものである。

例. ファイルパーミッション (rwX) のフラグ

```
>>> from enum import Flag      Enter    ←モジュールの読み込み
>>> class FilePermission(Flag): Enter    ←ファイルパーミッション用のクラス定義の開始
...     R = 4                  Enter
...     W = 2                  Enter
...     X = 1                  Enter
... Enter                      ←定義の記述の終了
>>> FilePermission            Enter    ←定義の確認
<flag 'FilePermission'>      ← flag 型
```

このフラグ `FilePermission` の各要素がビットフィールド中の各ビットになっていることが次の例で確認できる。

例. `FilePermission` の各要素を 2 進数として確認する (先の例の続き)

```
>>> bin( FilePermission.R.value ) Enter
'0b100'
>>> bin( FilePermission.W.value ) Enter
'0b10'
>>> bin( FilePermission.X.value ) Enter
'0b1'
```

Flag の要素はビット論理演算が可能である。

例. ビット論理和によるフィールドの重ね合わせ (先の例の続き)

```
>>> full = FilePermission.R | FilePermission.W | FilePermission.X Enter
>>> full Enter    ←全ての要素を重ね合わせたものは
<FilePermission.R|W|X: 7>    ←このようになる
>>> bin( full.value ) Enter    ←ビットフィールドにして確認
'0b111'    ←このようなビットフィールド
```

Flag で作成したビットフィールドは `in` 演算子で検査することができる。

例. `in` によるビットの検査 (先の例の続き)

```
>>> FilePermission.R in full Enter    ←読み込みビット (R) が full のフィールドで有効か?
True    ←有効である
```

複数のビットの状態を `in` で検査することもできる。

例. 複数のビットを `in` で検査する (先の例の続き)

```
>>> rw = FilePermission.R | FilePermission.W Enter    ←RW が有効なビットフィールド
>>> rw Enter    ←確認
<FilePermission.R|W: 6>
>>> rw in full Enter    ←このような検査が
True    ←可能である
>>> FilePermission.X in rw Enter    ←当然これも
False    ←可能である
```

課題. 次の実行結果が得られる理由について考察せよ.

in による検査の例. (先の例の続き)

```
>>> rx = FilePermission.R | FilePermission.X Enter
>>> rx in rw Enter
False ←この結果について考察せよ
```

注意) 上の課題のように, in 演算子によるビットフィールドの検査は誤解を生じやすく可読性に問題がある. 従って, ビットフィールドの検査には通常のビット論理演算子「&」、「|」、「^」、「~」などを用いることが推奨される.

4.18.5.1 Flag の要素への auto 関数による値の割り当て

Flag オブジェクトの要素に auto 関数で値を割り当てると, 順番に 2^0 , 2^1 , 2^2 , ... となる.

例. Flag の要素の値を auto 関数で割り当てる

```
>>> from enum import Flag, auto Enter ←モジュールの読み込み
>>> class FilePermission(Flag): Enter ←ファイルパーミッション用のクラス定義の開始
...     X = auto() Enter
...     W = auto() Enter
...     R = auto() Enter
...     FULL = R | W | X Enter
... Enter ←定義の記述の終了
```

各要素のビット状態を確認する.

例. 要素のビット状態の確認 (先の例の続き)

```
>>> bin( FilePermission.R.value ) Enter
'0b100' ←  $2^2$ 
>>> bin( FilePermission.W.value ) Enter
'0b10' ←  $2^1$ 
>>> bin( FilePermission.X.value ) Enter
'0b1' ←  $2^0$ 
>>> bin( FilePermission.FULL.value ) Enter
'0b111' ←上記3つの要素のビット論理和
```

4.18.5.2 整数の性質を持つビットフラグ: IntFlag

Flag オブジェクトは整数としての性質を持っていない. (次の例)

例. Flag オブジェクトに対する整数演算の試み (先の例の続き)

```
>>> FilePermission.R + 1 Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    FilePermission.R + 1
    ~~~~~
TypeError: unsupported operand type(s) for +: 'FilePermission' and 'int'
```

このように, 加算の演算「+」が定義されていない旨のエラーが発生する.

整数としての性質を持つビットフラグとして enum.IntFlag が提供されている. これを用いて先と同様のビットフラグを実装する例を次に示す.

例. IntFlag オブジェクトの定義

```
>>> from enum import IntFlag, auto Enter ←モジュールの読み込み
>>> class FPerm(IntFlag): Enter ←クラス定義の開始
...     X = auto() Enter
...     W = auto() Enter
...     R = auto() Enter
...     FULL = R | W | X Enter
... Enter ←定義の記述の終了
```

この FPerm の要素を確認する.

例. FPerm の要素の確認 (先の例の続き)

```
>>> FPerm.R  ←この要素は  
<FPerm.R: 4> ← FPerm 型である  
>>> FPerm.R + 1  ←整数としての加算が  
5 ←可能である
```

整数演算が可能だけでなく、簡単な記述 (value 属性を明に参照しない) が可能になる.

例. ビットフィールドとして表示 (先の例の続き)

```
>>> bin( FPerm.R )   
'0b100'  
>>> bin( FPerm.W )   
'0b10'  
>>> bin( FPerm.X )   
'0b1'  
>>> bin( FPerm.FULL )   
'0b111'
```

4.19 例外（エラー）の処理

「2.5.1.6 例外処理」(p.63) では、プログラムの実行中に発生するエラーや例外を扱うための方法を解説した。try～except の例外処理を適切に記述することで、例外が発生する場合においても処理系を中断することなくプログラムの実行を続けることができるが、ここでは更に進んだ内容について解説する。

4.19.1 エラーメッセージをデータとして取得する方法： traceback モジュール

エラーや例外の際に得られるメッセージを扱うための traceback モジュールを紹介する。このモジュールを使用するには次のようにしてモジュールを読み込む。

```
import traceback
```

この後、try～except の例外処理において、format_exc 関数を呼び出すと、例外やエラーの発生においてシステムが生成するメッセージを文字列型データとして取得することができる。

format_exc を用いたサンプルプログラム error01.py を示す。

プログラム：error01.py

```
1 import traceback
2
3 try:
4     # エラーが発生する部分
5     a = 2 * b
6 except:
7     # エラー処理の部分
8     print('*** 例外が発生しました ***')
9     er = traceback.format_exc()
10    print(er.rstrip())
11    print('*****\n')
12
13 print('--- プログラム終了 ---')
14 print('処理系の動作は中断されていません。')
```

解説

プログラムの5行目で、値が割当てられていない変数（記号）b が参照されている。通常ならばこの行はエラーとなり、処理系の実行は中断するが、例外処理が施されているので、8行目以降に処理が移行する。9行目に

```
er = traceback.format_exc()
```

という記述があり、これによりエラーメッセージが文字列型のデータとして変数 er に格納される。

このプログラムを実行した例を次に示す。

```
*** 例外が発生しました ***
Traceback (most recent call last):
  File "C:\Users\katsu\error01.py", line 6, in <module>
    a = 2 * b
    ^
NameError: name 'b' is not defined
*****

--- プログラム終了 ---
処理系の動作は中断されていません。
```

エラーによるプログラムの中断はあく、エラーメッセージが文字列のデータとして得られていることがわかる。

エラーメッセージだけでなく、例外事象に関する様々な情報を保持する**例外オブジェクト**を取得する方法について、後の「4.19.3 例外オブジェクト」(p.338) で解説する。

4.19.2 例外を発生させる方法

プログラム中の任意の行において設定された条件を検査し、その結果によって例外を発生させる assert 文がある。これは、プログラムのデバッグ作業において役立つことがある。

次のサンプルプログラム assert01.py について考える。

プログラム：assert01.py

```
1 # coding: utf-8
2 while True:
3     s = input('数値を入力>')
4     if s.isdigit():
5         n = int(s)
6         assert n%3!=0, '{}は3の倍数です.'.format(n)
```

このプログラムは、input 関数によってキーボード（標準入力）から文字列を読み込む処理を際限なく繰り返す形になっている。ただし「3の倍数」を表現する文字列を入力すると assert 文（6行目）により例外が発生してプログラムの実行が中断する。（次の例参照）

assert1.py の実行例.

```
数値を入力>a Enter
数値を入力>10 Enter
数値を入力>20 Enter
数値を入力>33 Enter

Traceback (most recent call last):
  File "C:\¥Users¥katsu¥assert01.py", line 6, in <module>
    assert n%3!=0, '{}は3の倍数です.'.format(n)
    ~~~~~
AssertionError: 33は3の倍数です.
```

● assert 文

書き方： `assert 条件式, エラーメッセージ`

「条件式」が真（True）とならない場合（False となる場合）に「エラーメッセージ」を生成して例外 `AssertionError` を発生させる。

assert 文とは別に、任意の種類の例外（エラー）を発生させる `raise` 文も使用できる。（次の例参照）

例. raise 文で意図的に ValueError を起こす

```
>>> raise ValueError('故意に生成した ValueError') Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 故意に生成した ValueError
```

←エラーメッセージが表示される

● raise 文

書き方： `raise 例外の名前 (エラーメッセージ)`

「例外の名前」の例外（エラー）を発生させる。このとき「エラーメッセージ」をメッセージとする。

assert, raise 文で発生する例外は try 文でハンドリングすることができる。

4.19.3 例外オブジェクト

Python 処理系において例外（エラー）が発生すると例外オブジェクトが作成される。これに関して次のサンプルプログラム `error02.py` を用いて考える。

プログラム：error02.py

```
1 # coding: utf-8
2 try:
3     a = 1/0
4 except ZeroDivisionError as e:
5     print('例外オブジェクト:', e)
6     print('クラス:', type(e))
```

このプログラムは3行目で0による除算を試みており、これによって `ZeroDivisionError` という例外が発生するが、この例外の事象（関連情報）が例外オブジェクトとして変数 `e` に渡されて（4行目）いる。その内容とクラスに関する情報を5～6行目で出力している。このプログラムを実行すると次のような出力が得られる。

出力の例.

```
例外オブジェクト: division by zero
クラス: <class 'ZeroDivisionError'>
```

このプログラムのように、エラーハンドリングの際に

```
except 例外の種類 as 変数
```

と記述することで、発生した例外オブジェクトが「変数」に渡される。また「例外の種類」の部分は、例外オブジェクトのクラス名である。

4.19.3.1 例外オブジェクトのクラス階層

システムに発生する例外の種類は、例外オブジェクトのクラス階層として分類されており、すべての例外の基底クラスは `BaseException` である。またこのクラス直下の派生クラスに `Exception` クラスがあり、これの派生クラスとして凡そ全ての例外や警告のクラスが含まれる。例えば先のプログラム `error02.py` でハンドリングした `ZeroDivisionError` はクラス階層内で

```
BaseException → Exception → ArithmeticError → ZeroDivisionError
```

と位置づけられている。従って、例外をハンドリングする際の `except` の部分を

```
except Exception as e:
```

などと記述することで、様々な種類のエラーを一括して扱うことができる。

参考)

後で紹介するプログラム `getSubclasses.py` (p.363) を用いて全ての例外オブジェクトの階層関係を出力して観察されたい。

4.20 使用されているシンボルの調査

値の設定されている変数やオブジェクトの名前（シンボル）を調べる方法について説明する。グローバルのスコープで定義されているシンボル²²² を調べるには `globals` 関数を使用する。(次の例参照)

例. グローバルのシンボルを調べる²²³

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__spec__': None, '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>}
```

← シンボルへの値の
割当てが辞書の形で
得られる

この結果と `in` 演算子を用いることで、あるシンボルが定義済みか未定義かを判定することができる。例えば、`x` というシンボルに何も割当てられていない状況でそれを検証する例を次に示す。

例. シンボルの使用状態を調べる

```
>>> 'x' in globals()
False
>>> x = 2
>>> 'x' in globals()
True
```

← シンボル 'x' に値が定義されているかどうかを調べる。
← 未定義であることがわかる
← シンボル 'x' に値を割り当てる
← 割当て状況を再度調べる。
← 値を持つシンボルであることがわかる

関数の内部のスコープ（ローカルのスコープ）で使用されているシンボルを調べるには `locals` 関数を使用する。

文字列として記述されたプログラムを実行する関数²²⁴ において、グローバル変数の名前空間を使用する場合がある。そのような場合に `globals()` 関数が返す辞書が役立つ。`globals()` が返す辞書には `__name__`, `__builtins__`, `__package__` といった重要なエントリが含まれており、真のグローバルの名前空間となる。

²²² グローバル変数（大域変数）など。

²²³ 実行するシステム毎に結果が異なる。

²²⁴ `exec` や `eval`、あるいは `timeit` モジュールの時間計測用のメソッドなどがそれに該当する。

4.21 with 構文

実用的なアプリケーションプログラムを開発する際には、情報処理のための主たるアルゴリズムの実装だけにとどまらず、アクセスするシステム資源の準備処理や終了処理、あるいは例外処理といった細かい点に関する配慮が求められる。例えば、ファイルに対する入出力の処理においては、

- ファイルのオープン
- ファイルに対する入出力とそれに関連する例外処理 (try～except～) のハンドリング
- ファイルのクローズ

といったことを実行することになるが、上記強調下線部の処理は主たるアルゴリズムの実装に付随する部分と考えることができ、その部分を切り離して記述することで主たるプログラムの部分を簡潔に表現し、可読性を高めることができる場合がある。ここではファイルへのアクセスを例に挙げて with 構文の使用方法について説明する。

次に示すプログラム fileWith01.1.py は、ファイル exfile.txt の内容を読み込んで表示するものである。

プログラム：fileWith01.1.py

```
1 # coding: utf-8
2 fname = 'exfile.txt'
3
4 f = open( fname, 'r', encoding='utf-8' )
5 m = f.read()
6 f.close()
7
8 print( m )
```

入力用のファイルとして次のような内容のものを用意してプログラムの実行を試みる。

入力用ファイル：exfile.txt

```
1 exfile.txtの中身
```

プログラムの実行結果は次のようになる。

例. Windows 環境での実行

```
C:\¥Users¥katsu> py fileWith01.1.py  [Enter]  ←コマンドからスクリプトを実行
exfile.txtの中身                      ←結果表示
```

次に、同様の処理を実現するプログラムを with 構文で実装したものを fileWith01.2.py に示す。

プログラム：fileWith01.2.py

```
1 # coding: utf-8
2 fname = 'exfile.txt'
3
4 with open( fname, 'r', encoding='utf-8' ) as f:
5     m = f.read()
6
7 print( m )
```

【with 構文の書き方】

with 処理対象のオブジェクトを生成する記述 as 生成されたオブジェクト
(生成されたオブジェクトに対する処理)

プログラム fileWith01.2.py では with 構文を用いることでファイルをクローズする処理が省略されているのがわかるが、この例では with 構文の利便性を理解するには不十分である。

次に、同様の機能を持つプログラムを例外処理までを含んだ形で実装する方法について考える。fileWith01.2.py の 4 行目に、存在しないファイル 'nofile.txt' を与えた形に書き換えた fileWith02.1.py を次に示す。

プログラム：fileWith02.1.py

```
1 # coding: utf-8
2 fname = 'nofile.txt'          # 存在しないファイル nofile.txt
3
4 with open( fname, 'r', encoding='utf-8' ) as f:
5     m = f.read()
6
7 print( '読み取り結果:',m )
```

このプログラムを実行すると、ファイルが存在しない旨の次のようなエラーメッセージが表示される。

例. ファイルが存在しない旨のエラー

C:\Users\katsu> py fileWith02.1.py ←コマンドからスクリプトを実行

```
Traceback (most recent call last):  ←エラーメッセージ
  File "C:\Users\katsu\fileWith02.1.py", line 4, in <module>
    with open( fname, 'r', encoding='utf-8' ) as f:
    ~~~~~
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

‘FileNotFoundError’が発生しており、プログラムの4行目で実行が中断されている。次に、try～except～構文によって例外処理をハンドリングした、プログラムの実行が中断しない形の実装について考える。

with 構文に与えるオブジェクトは、with 構文に対応したメソッドを備えていなければならない。具体的には、with 構文によって処理を開始する時点で呼び出される `__enter__` メソッド（初期化処理）と、with 構文による処理を終了する時点で呼び出される `__exit__` メソッド（終了処理）である。これらのメソッドを実装したクラスを定義して、そのクラスのインスタンスを with 構文で生成して使用する形にする。これを実現したプログラム例 fileWith02.2.py を示す。

プログラム：fileWith02.2.py

```
1 # coding: utf-8
2 #--- ファイルを扱う独自のクラス ---
3 class MyFileReader:
4     #--- コンストラクタ ---
5     def __init__(self,fname):
6         self.fn = fname
7         self.f = None
8     #--- 開始処理 ---
9     def __enter__(self):
10         try:
11             self.f = open( self.fn, 'r', encoding='utf-8' )
12             print( 'ファイル',self.fn,'をオープンしました。' )
13         except:
14             self.f = None
15             print( 'ファイル',self.fn,'をオープンできません。' )
16         return self
17     #--- ファイルの読み込み ---
18     def read(self):
19         if self.f:
20             try:
21                 r = self.f.read()
22                 print( 'ファイルからデータを読み込みました。' )
23             except:
24                 r = None
25                 print( 'ファイルから読み込むことができません。' )
26         else:
27             print( 'ファイルがオープンされていません。' )
28             r = None
29         return r
30     #--- 終了処理 ---
31     def __exit__(self,et,ev,tr):
32         if self.f:
33             self.f.close()
34             print( 'ファイルをクローズしました。' )
35         else:
36             print( '処理不可能で終了します。' )
37         return True
38
```

```

39 #--- 実行部分 ---
40 fname = 'nofile.txt'
41
42 with MyFileReader( fname ) as f:
43     m = f.read()
44
45 print( '読み取り結果:',m )

```

解説

3行目から定義が始まる `MyFileReader` クラスはファイルのオープンと読み込みに伴う例外処理をハンドリングするためのもので、コンストラクタの引数には対象のファイル名を与えてインスタンスを生成する。このクラスのオブジェクトは `with` 構文による処理の開始において `__enter__` メソッドが実行されて対象のファイルがオープンされる。また、このクラスには `read` メソッドが定義されており、これによりファイルの内容が読み込まれる。 `with` 構文が終了する際には `__exit__` メソッドが呼び出されてファイルがクローズされる。

`with` 構文を使わない形で例外処理まで含めた処理を実現すると、ファイルをオープンする記述とクローズする記述、更に、それらに関する例外処理のハンドリング、入出力処理に関する例外処理のハンドリングの記述を主たる処理の中に散りばめることになり、プログラム全体の見通しが悪くなる。それに対して、この実装の42行目以降を見ると、入出力の主たる処理のみが簡潔に記述されていることがわかる。

このプログラムを実行すると次のように表示される。

例. `fileWith02_2.py` の実行

```

C:\Users\katsu> py fileWith02_2.py Enter ←コマンドからスクリプトを実行
ファイル nofile.txt をオープンできません。 ←例外のハンドリングによるメッセージ出力
ファイルがオープンされていません。
処理不可能で終了します。
読み取り結果: None

```

例外事象に対してもプログラムを中断すること無く、プログラムの動作によってメッセージを表示している。また、プログラムの40行目で入力用のファイル名として先に示したファイル `'exfile.txt'` を与えて実行すると次のような表示となる。

例. 入力ファイルを変えて `fileWith02_2.py` を実行

```

C:\Users\katsu> py fileWith02_2.py Enter ←コマンドからスクリプトを実行
ファイル exfile.txt をオープンしました。 ←プログラムによるメッセージ出力
ファイルからデータを読み込みました。
ファイルをクローズしました。
読み取り結果: exfile.txt の中身

```

ファイルのオープンと内容の読み取りができていたことがわかる。今回のプログラム `fileWith02_2.py` では、`__enter__` メソッドと `read` メソッドの中で例外処理のハンドリング (`try~except~` の記述) をしている。

次に、`read` メソッドによる読み込み時におけるエラーハンドリングをせずに `__exit__` に例外のハンドリングを委ねる例 (プログラム `fileWith02_3.py`) について考える。

プログラム: `fileWith02_3.py`

```

1  # coding: utf-8
2  #--- ファイルを扱う独自のクラス ---
3  class MyFileReader:
4      #--- コンストラクタ: ファイルオープン ---
5      def __init__(self, fname):
6          self.fn = fname
7          self.f = None
8      #--- 開始処理 ---
9      def __enter__(self):
10         try:
11             self.f = open( self.fn, 'r', encoding='utf-8' )
12             print( 'ファイル',self.fn,'をオープンしました。' )
13         except:
14             self.f = None
15             print( 'ファイル',self.fn,'をオープンできません。' )
16         return self

```

```

17     #--- ファイルの読み込み ---
18     def read(self):
19         return self.f.read()
20     #--- 終了／例外処理 ---
21     def __exit__(self,et,ev,tr):
22         if self.f:
23             self.f.close()
24             print( '例外のタイプ:\t',et )
25             print( '例外の値:\t',ev )
26             print( 'トレース:\t',tr )
27             return True
28
29 #--- 実行部分 ---
30 fname = 'nofile.txt'
31
32 with MyFileReader( fname ) as f:
33     m = f.read()
34
35 if 'm' in globals():          # 変数 m が値を持っておれば出力
36     print( '読み取り結果:',m )

```

read メソッドの呼び出しにおいて例外処理のハンドリングをしない場合、例外発生時に `__exit__` メソッドが呼び出されて with 構文が終了する。(次の例参照)

例. fileWith02_3.py の実行

```

C:\Users\katsu> py fileWith02_3.py  Enter    ←コマンドからスクリプトを実行
ファイル nofile.txt をオープンできません。    ←__enter__ メソッドによる例外処理
例外のタイプ:    <class 'AttributeError'>    ←__exit__ メソッドによる例外処理
例外の値:        'NoneType' object has no attribute 'read'
トレース:        <traceback object at 0x000001A1F6CD5348>

```

この実行例では、エラータイプ、エラーの値、トレース情報が `__exit__` メソッドの引数 `et, ev, tr` にそれぞれ渡されている。実際のアプリケーションプログラムにおいては、これらの値を用いて更に高度な例外処理を実現すると良い。また、例外が発生せずに正常に with 構文を終了する場合は、これらの引数には `None` が設定される。試みとして、fileWith02_3.py の 30 行目で、実際に存在するファイル名 'exfile.txt' を与えてみるとそれが確認できる。(次の例参照)

例. 入力ファイルを変えて fileWith02_3.py を実行

```

C:\Users\katsu> py fileWith02_3.py  Enter    ←コマンドからスクリプトを実行
例外のタイプ:    None    ← None になっている
例外の値:        None    ← None になっている
トレース:        None    ← None になっている
読み取り結果:    exfile.txt の中身    ←読み取った内容

```

ここでは、ファイルのオープンと内容の読み取りを例に挙げて with 構文について説明したが、通信を始めとする例外事象の発生しやすい資源へのアクセスに関する処理においても with 構文は見やすいプログラムの実現に役立つことが多い。

【with 構文のまとめ】

次のような処理を実現する際に with 構文によってプログラムを見やすく記述することができる可能性がある。

- ・ 初期化と終了の処理が伴う処理
- ・ 例外処理が必要となる処理

また、with 構文を使用する場合は、そのためのオブジェクトのクラスを定義し、上記のような処理はクラス内に隠蔽し、呼び出し元では意識しないように工夫する。

4.22 デコレータ

Python では、関数の定義を装飾するためのデコレータを使用することができる。次に示すサンプルプログラム decorator01.py はデコレータを使用する例であり、まずはこれに沿って説明する。

プログラム：decorator01.py

```
1  # coding: utf-8
2  #--- デコレータの定義 -----
3  def dcr1(f):      # 関数オブジェクトを引数に取る関数
4      # 実際に実行する関数
5      def retf(a):
6          r = f(a)
7          return '<p>' + str(r) + '</p>'
8      # デコレータが返す関数オブジェクト
9      return retf
10
11 #--- デコレータの関数定義への適用 -----
12 @dcr1
13 def mul2(n):      # 受け取った数を2倍する関数
14     return 2*n
15
16 @dcr1
17 def pow2(n):      # 受け取った数を2乗する関数
18     return n**2
19
20 #--- 実行 -----
21 m = mul2(4)
22 print( '戻り値:', m, '型:', type(m) )
23
24 m = pow2(3)
25 print( '戻り値:', m, '型:', type(m) )
```

このプログラムでは、与えられた数を2倍する関数 mul2 と、与えられた数の2乗を求める関数 pow2 が定義されている。一見するとこれら関数は、数の型（整数、浮動小数点数）の値を返すものと思われるが、このプログラムをスクリプトとして実行すると次のような結果となる。

decorator01.py の実行結果：

```
戻り値:  <p>8</p> 型:  <class 'str'>
戻り値:  <p>9</p> 型:  <class 'str'>
```

関数の戻り値は、計算結果を '

…

' で括った文字列となっていることがわかる。

解説：

プログラムの3～9行目に定義されている関数 dcr1 は、仮引数 f に受け取った関数オブジェクトを使用して、別の関数（内部関数 retf）を定義し、その関数オブジェクト retf を返すものである。そして、関数 mul2, pow2 の定義の直前に、アットマーク「@」を用いる形で関数 dcr1 をデコレータとして記述している。

デコレータ @dcr1 で装飾された mul2, pow2 の関数定義は、デコレータ関数の内部関数 retf によってその定義が変更される。従ってデコレータは、対象とする関数の定義に一定の変更（装飾）を加える仕組みであると言える。

デコレータの存在意義：

Python のデコレータはプログラミングの概念としては本質的なものではなく、デコレータを用いなくてもアプリケーション開発を十分に行うことができるが、これを用いることでプログラムの可読性を高めることができる場合がある。

【理解を深めるための別の例】

先に上げたプログラム decorator01.py と同じ働きをする別のプログラム decorator02.py を次に示す。

プログラム：decorator02.py

```
1  # coding: utf-8
2  #--- デコレータの定義 -----
```

```

3 def dcr1(f):      # 関数オブジェクトを引数に取る関数
4     # 実際に行う関数
5     def retf(a):
6         r = f(a)
7         return '<p>' + str(r) + '</p>'
8     # デコレータが返す関数オブジェクト
9     return retf
10
11 #--- デコレータの適用対象の関数 -----
12 def mul2(n):      # 受け取った数を2倍する関数
13     return 2*n
14
15 def pow2(n):      # 受け取った数を2乗する関数
16     return n**2
17
18 #--- 実行 -----
19 mul2 = dcr1( mul2 )      # デコレータ関数による定義の変更
20 m = mul2(4)
21 print( '戻り値:', m, '型:', type(m) )
22
23 pow2 = dcr1( pow2 )      # デコレータ関数による定義の変更
24 m = pow2(3)
25 print( '戻り値:', m, '型:', type(m) )

```

このプログラムでは、装飾用の関数 `dcr1` をデコレータとしてではなく単なる関数として使用し、関数 `mul2`, `pow2` を別の関数として再定義して (19, 23 行目) いる。このプログラムを実行すると、先の `decorator01.py` と同じ結果となる。これら2つのプログラムを比較することで、デコレータの働きを確認することができる。

課題. 先の `decorator01.py` に次のような関数定義を加え、その関数の評価結果を確認せよ。(デコレータの入れ子)

```

@dcr1
@dcr1
def add1(n):
    return n+1

```

4.22.1 引数を取るデコレータ

引数を取るデコレータを定義することが可能であり、引数に受け取った値を用いて他の関数を装飾することができる。次のサンプルプログラム `decorator03.py` でそれを示す。

プログラム: `decorator03.py`

```

1  # coding: utf-8
2  #--- デコレータの定義 -----
3  def dcr2( tg ): # 引数を取るデコレータ
4      # 実際に行う関数
5      def retf( f ):      # 関数オブジェクトを引数に取る関数
6          def wkf( a ):    # 実際に行う関数
7              tg1 = '<' + tg + '>'
8              tg2 = '</' + tg + '>'
9              r = f(a)
10             return tg1 + str(r) + tg2
11         return wkf      # 関数オブジェクトを返す
12     # デコレータが返す関数オブジェクト
13     return retf
14
15 #--- デコレータの関数定義への適用 -----
16 @dcr2('p')      # 引数 'p' を取るデコレータ
17 def mul2(n):      # 受け取った数を2倍する関数
18     return 2*n
19
20 @dcr2('body')    # 引数 'body' を取るデコレータ
21 def pow2(n):      # 受け取った数を2乗する関数
22     return n**2
23
24 @dcr2('html')    # 引数 'html' を取るデコレータ
25 @dcr2('body')    # 引数 'body' を取るデコレータ

```



```

26 @dcr2('p')          # 引数 'p' を取るデコレータ
27 def html(s):         # 引数をそのまま返す関数
28     return s
29
30 #--- 実行 -----
31 m = mul2(1)
32 print( '戻り値:', m, '型:', type(m) )
33
34 m = pow2(3)
35 print( '戻り値:', m, '型:', type(m) )
36
37 m = html('テキストメッセージ')
38 print( '戻り値:', m, '型:', type(m) )

```

この例のように、デコレータ関数の内部関数を2重に構成することで、引数を取るデコレータを定義することができる。このプログラムを実行すると次のような結果となる。

decorator03.py の実行結果：

```

戻り値:  <p>2</p> 型:  <class 'str'>
戻り値:  <body>9</body> 型:  <class 'str'>
戻り値:  <html><body><p>テキストメッセージ</p></body></html> 型:  <class 'str'>

```

解説：

デコレータ関数 dcr2 の引数はデコレータとして記述する際に与える引数である。最初の内部関数 retf は引数に関数オブジェクトを取り、これが装飾対象の関数に対応する。更に内側の内部関数 wkf は装飾された関数の呼び出しに対応するものであり、wkf の引数に与えたものが装飾対象の関数に渡される。

24～28 行目に定義された関数 html はデコレータによる装飾が入れ子になっており、結果として HTML の形式で戻り値が得られている。

4.23 データ構造の整形表示： pprint モジュール

pprint モジュールを用いると、長いデータ構造を整形表示することができる。具体的には pprint モジュールの pprint 関数を使用する。

例. 長いリストの表示

```

>>> lst = [[0]*10]*5  Enter      ←長いリストの作成
>>> print( lst )      Enter      ← print 関数で表示する
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

```

このように、長いリストを print 関数で表示するとディスプレイの右端で折り返して表示される。このような場合、pprint モジュールの pprint 関数を使用すると次のように見やすく表示される。

例. pprint 関数による整形表示（先の例の続き）

```

>>> from pprint import pprint  Enter      ← pprint モジュールから pprint 関数を読み込む
>>> pprint( lst )             Enter      ← pprint 関数で表示する
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

```

先の例よりも見やすくなっている。

pprint は辞書型オブジェクトも整形表示する。

例. 辞書オブジェクトの整形表示 (先の例の続き)

```
>>> dic = {'apple': 'りんご', 'orange': 'みかん', 'lemon': 'レモン',  
           'ぶどう': 'grape', '梨': 'pear'} Enter ←辞書オブジェクトの作成  
>>> pprint( dic, width=70 ) Enter ←幅を指定して整形表示  
{'apple': 'りんご',  
  'lemon': 'レモン',  
  'orange': 'みかん',  
  'ぶどう': 'grape',  
  '梨': 'pear'}
```

キーワード引数 'width=' には幅 (表示桁数) を与え, 表示対象がこの範囲に収まらない場合に辞書の各要素を 1 行ずつ表示する。

pprint の制御のための引数には様々なものがあり, 詳しくは Python の公式インターネットサイトなどを参照のこと。

4.24 文字列の整形処理 (分割, 折り返しなど): textwrap モジュール

Python の標準ライブラリである textwrap モジュールを用いると, 文字列に対して分割や折り返しをはじめとする整形処理ができる。このモジュールは使用に先立って,

```
import textwrap
```

などとして Python 言語処理系に読み込む必要がある。以下の解説における実行例では, 当該モジュールを読み込んだ状態を前提とする。

4.24.1 長い文字列の分割

wrap, fill を用いると, 指定した長さで文字列を分割することができる。

書き方: wrap(対象文字列, width=長さ)

書き方: fill(対象文字列, width=長さ)

「対象文字列」を「長さ」毎に分割したものを返す。wrap は結果をリストで, fill は結果を 1 つの文字列として返す。

例. wrap 関数

```
>>> import textwrap Enter ←モジュールの読み込み  
>>> s = 'あいうえおかきくけこさしすせそたちつてと' Enter ←この文字列を  
>>> textwrap.wrap( s, width=5 ) Enter ←長さ 5 で分割  
['あいうえお', 'かきくけこ', 'さしすせそ', 'たちつてと'] ←分割結果のリスト  
>>> textwrap.wrap( s, width=10 ) Enter ←長さ 10 で分割  
['あいうえおかきくけこ', 'さしすせそたちつてと'] ←分割結果のリスト
```

例. fill 関数 (先の例の続き)

```
>>> print( textwrap.fill( s, width=5 ) ) Enter ←長さ 5 で分割  
あいうえお ←分割結果のテキスト (ここから)  
かきくけこ  
さしすせそ  
たちつてと ←分割結果のテキスト (ここまで)  
>>> print( textwrap.fill( s, width=10 ) ) Enter ←長さ 10 で分割  
あいうえおかきくけこ ←分割結果のテキスト (ここから)  
さしすせそたちつてと ←分割結果のテキスト (ここまで)
```

fill 関数が返す値は, 改行のエスケープシーケンス '\n' を含む 1 つの文字列である。(次の例)

例. fill 関数の戻り値 (先の例の続き)

```
>>> textwrap.fill( s, width=5 ) Enter ←上と同じ処理を print を使わずに実行  
'あいうえお\nかきくけこ\nさしすせそ\nたちつてと' ←処理結果
```

4.24.2 インデントの除去

Python では三重の引用符を用いて複数行の文字列を記述することができるが、余計なインデントができてしまうことがある。(次の例)

例. 余計なインデントを持つ文字列

```
>>> s = '''      Enter      ←複数行に渡る文字列の記述（ここから）
...     一行目      Enter
...     二行目      Enter
...     三行目      Enter
...     '''      Enter      ←複数行に渡る文字列の記述（ここまで）
>>> s      Enter      ←内容確認
'¥n     一行目¥n     二行目¥n     三行目¥n     '      ←インデントの空白を含んでいる
```

dedent 関数を用いると、各行の先頭に共通する桁数の空白文字がある場合にそれらを除去することができる。(次の例)

例. dedent によるインデントの除去 (先の例の続き)

```
>>> s2 = textwrap.dedent( s )      Enter      ←インデント除去処理
>>> s2      Enter      ←除去結果の確認
'¥n 一行目¥n 二行目¥n 三行目¥n'      ←除去結果
```

4.24.3 インデントの挿入

indent 関数を使用することで、改行のエスケープシーケンス「¥n」によって構成された文字列にインデントを挿入することができる。

書き方: indent(対象文字列, prefix=インデント文字列)

「インデント文字列」を「対象文字列」内の各行に挿入する。

例. インデントの挿入 (先の例の続き)

```
>>> s3 = textwrap.indent( s2, prefix='--> ' )      Enter      ←各行の先頭に '--> ' を挿入する
>>> s3      Enter      ←挿入処理結果の確認
'¥n--> 一行目¥n--> 二行目¥n--> 三行目¥n'      ←挿入処理結果
>>> print( s3 )      Enter      ←整形表示による確認
--> 一行目      ←ここから
--> 二行目
--> 三行目
                        ←ここまで
```

上の例では、先頭行と最終行に「¥n」があるので処理結果の表示が少し不自然に見えるかもしれない。この場合は、strip メソッドを組み合わせるなどして更に整形すると良い。(次の例)

例. 上の例の改善 (先の例の続き)

```
>>> print( textwrap.indent( s2.strip(), prefix='--> ' ) )      Enter      ←更なる工夫
--> 一行目      ←整形結果（ここから）
--> 二行目
--> 三行目      ←整形結果（ここまで）
```

4.25 処理環境に関する情報の取得

1つのPythonのアプリケーションプログラムを異なる処理環境で実行する²²⁵ 場合、当該アプリケーションが実行中の処理環境に関する情報を取得して、プログラムの実行方法を変えなければならないケースが出てくる。ここでは、実行中の処理環境に関する情報を取得するためのいくつかの方法を紹介する。

4.25.1 Pythonのバージョン情報の取得

sys モジュールを使用すると実行中のPython処理系のバージョン情報を取得することができる。1つの方法は、次の例に示すような version プロパティの参照である。

例. Python 処理系のバージョン情報の取得（その1）

```
>>> import sys      Enter      ← sys モジュールの読み込み
>>> sys.version     Enter      ← version プロパティの参照
'3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)]'
                                     ↑ バージョン情報が表示されている
```

もう1つの方法は、version_info プロパティの参照で、バージョンに関する各種の情報が namedtuple ²²⁶ として得られる。（次の例を参照）

例. version_info プロパティ（先の例の続き）

```
>>> v = sys.version_info Enter      ← version_info プロパティの参照
>>> print( v )      Enter      ← version_info プロパティの表示
sys.version_info(major=3, minor=11, micro=2, releaselevel='final', serial=0) ← 結果表示
>>> v.major         Enter      ← 主バージョンのプロパティ 'major' の参照
3                    ← 結果表示
```

4.25.2 platform モジュールの利用

Python 処理系に標準で添付されている platform モジュールを利用すると、処理環境に関する詳しい情報を得ることができる。

例. OS の詳細な版を調べる platform 関数

```
>>> import platform Enter      ←モジュールの読み込み
>>> platform.platform() Enter   ← platform 関数の実行
'Windows-10-10.0.22621-SP0'    ← 結果表示
```

platform モジュールが提供する関数の1部を表 43 に示す。

表 43: platform モジュールの関数の一部

関数	戻り値
platform.architecture()	アーキテクチャのビット数と実行可能ファイルのリンク形式のタプル
platform.processor()	プロセッサ名
platform.platform()	プラットフォーム名
platform.system()	システム名 (OS 名). 'Linux', 'Windows', 'Darwin' (macOS の場合)
platform.version()	システムのリリース情報
platform.mac_ver()	Apple 社の Macintosh の OS に関する情報 (Windows 環境では無意味)
platform.python_compiler()	Python 処理系をビルドしたコンパイラに関する情報

処理環境の各種情報を表示するプログラム例を platform01.py に示す。

²²⁵クロスプラットフォームに対応させるという意味。
²²⁶「4.16.3 namedtuple」(p.325) で解説している。

プログラム：platform01.py

```
1 # coding: utf-8
2 import platform
3 import sys
4
5 print('sys.version:', sys.version, '\n')
6
7 print('platform.architecture()\t:', platform.architecture())
8 print('platform.processor()\t:', platform.processor())
9 print('platform.platform()\t:', platform.platform())
10 print('platform.system()\t:', platform.system())
11 print('platform.version()\t:', platform.version())
12 print('platform.mac_ver()\t:', platform.mac_ver())
13 print('platform.python_compiler():', platform.python_compiler())
```

このプログラムの実行例を次に示す。

例. Windows 10 での実行例

```
C:\Users\katsu> py platform01.py [Enter] ←実行開始
sys.version: 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)]
platform.architecture() : ('64bit', 'WindowsPE')
platform.processor()    : Intel64 Family 6 Model 140 Stepping 2, GenuineIntel
platform.platform()     : Windows-10-10.0.22621-SP0
platform.system()       : Windows
platform.version()      : 10.0.22621
platform.mac_ver()      : ('', ('', '', '')), '')
platform.python_compiler(): MSC v.1934 64 bit (AMD64)
```

例. macOS Catalina での実行例

```
katsu$ python3 platform01.py [Enter] ←実行開始
sys.version: 3.11.2 (v3.11.2:878ead1ac1, Feb 7 2023, 10:02:41) [Clang 13.0.0 (clang-1300.0.29.30)]
platform.architecture() : ('64bit', '')
platform.processor()    : i386
platform.platform()     : macOS-10.15.7-x86_64-i386-64bit
platform.system()       : Darwin
platform.version()      : Darwin Kernel Version 19.6.0: Tue Jun 21 21:18:39 PDT 2022;
                        root:xnu-6153.141.66~1/RELEASE_ARM_T8020
platform.mac_ver()      : ('10.15.7', ('', '', '')), 'x86_64')
platform.python_compiler(): Clang 13.0.0 (clang-1300.0.29.30)
```

4.25.3 実行中のプログラムの PID の取得

os モジュールの `getpid` 関数を用いると、実行中の当該プログラムのプロセス ID (PID)²²⁷ を取得することができる。

例. 実行中のプログラム (Python 処理系) の PID を調べる

```
>>> import os [Enter] ← os モジュールの読み込み
>>> os.getpid() [Enter] ← PID の調査
5624          ←得られた PID
```

PID は、Python 言語処理系が起動する際に実行環境の OS が与える。

4.25.4 環境変数の参照

os モジュールの `environ` には環境変数とその値が辞書の形で保持されている。すなわち、

`os.environ[環境変数の名前]`

として環境変数の値が参照できる。例えば、Windows 環境でユーザのホームドライブ、ホームディレクトリは環境変

²²⁷ 正確には、スクリプトを実行している Python 言語処理系 (インタプリタ) の PID。

数 'HOMEDRIVE', 'HOMEPATH' に保持されている²²⁸ が、この変数名をキーにして `os.environ` にアクセスすると値を参照することができる。(次の例)

例. Windows 環境でユーザのホームドライブとホームディレクトリを取得する

```
>>> import os  [Enter]    ←モジュールの読み込み
>>> homed = os.environ['HOMEDRIVE']  [Enter]    ←ホームドライブを取得
>>> homep = os.environ['HOMEPATH']  [Enter]    ←ホームディレクトリのパスを取得
```

例. ホームドライブとホームディレクトリを確認する(先の例の続き)

```
>>> print( homed )  [Enter]    ←ホームドライブを表示
C:                  ←ホームドライブ
>>> print( homep )  [Enter]    ←ホームディレクトリのパスを表示
¥Users¥katsu       ←ホームディレクトリのパス
>>> home = homed + homep  [Enter]    ←完全なホームディレクトリを作成
>>> print( home )  [Enter]    ←完全なホームディレクトリを表示
C:¥Users¥katsu     ←完全なホームディレクトリ
```

全ての環境変数の名前と値を表示するプログラムの例を `osenv01.py` に示す.

プログラム: `osenv01.py`

```
1  # coding: utf-8
2  import os
3
4  # 環境変数の取得
5  env = list( os.environ.keys() )    # 変数名のリストを取得
6  env.sort()                        # 変数名リストを整列
7
8  # 環境変数の値の表示
9  for e in env:
10     print( e, '=', os.environ[e] )
```

このプログラムを実行した例を次に示す.

実行例.

```
C:¥Users¥katsu> py osenv01.py  [Enter]    ←実行開始
ALLUSERSPROFILE = C:¥ProgramData
APPDATA = C:¥Users¥katsu¥AppData¥Roaming
COMMONPROGRAMFILES = C:¥Program Files¥Common Files
COMMONPROGRAMFILES(X86) = C:¥Program Files (x86)¥Common Files
COMMONPROGRAMW6432 = C:¥Program Files¥Common Files
:
(以下省略)
:
```

システムやプロセスの状態を詳細に調べるためのライブラリ `psutil` がサードパーティによって開発, 公開されている. これに関しては公式インターネットサイト (<https://github.com/giampaolo/psutil>) など²²⁹ を参照のこと.

²²⁸Mac, Linux ではユーザのホームディレクトリは環境変数 'HOME' に保持されている.

²²⁹拙書「Python3 ライブラリブック」でも解説しています.

4.26 ファイル、ディレクトリに対する操作：shutil モジュール

Python 処理系に標準的に添付されている shutil モジュールを使用すると、ファイルやディレクトリの複製や、圧縮ファイル、書庫ファイルの取り扱いができる。本書では shutil モジュールが提供するいくつかの基本的な機能について使用例を挙げて紹介する。shutil に関する詳しい事柄は Python の公式ドキュメント（公式インターネットサイトなど）を参照のこと。shutil を使用するには、

```
import shutil
```

としてモジュールを Python 処理系に読み込む。

4.26.1 ファイルの複製

shutil の copy 関数を使用するとファイルを複製することができる。

書き方： `shutil.copy(元のファイル, 複製ファイル)`

「元のファイル」には、元のファイルのファイル名（パス名）を、「複製ファイル」には複製ファイルのファイル名（パス名）を与える。

例。 ファイル 'TestFile1.txt' の複製 'TestFile1-2.txt' を作成する

```
>>> import shutil  [Enter]          ←モジュールの読み込み
>>> shutil.copy( 'TestFile1.txt', 'TestFile1-2.txt' ) [Enter]  ←複製処理
'TestFile1-2.txt'          ←複製のファイル名が返される
```

当然のことではあるが、copy 関数の実行において、元のファイルが存在していなければならない。元のファイルが存在していない場合は copy 関数実行時に FileNotFoundError というエラーが発生する。

- copy 関数の第 2 引数「複製ファイル」にディレクトリ名を与えると、元のファイルがそのディレクトリの下に同じファイル名で複製される。
- copy 関数は複製時にファイルの内容に加えて、パーミッションも複製する。ただし、複製ファイルのタイムスタンプなどの情報は copy 関数実行時のものとなる。タイムスタンプなどを含めた更に多くの管理情報を複製するには `copy2` 関数を使用する。

4.26.2 ディレクトリ階層の複製

shutil の copytree 関数を使用すると、ディレクトリをその内容ごと複製することができる。

書き方： `shutil.copytree(元のディレクトリ, 複製ディレクトリ)`

「元のディレクトリ」には、元のディレクトリの名前（パス名）を、「複製ディレクトリ」には複製ディレクトリの名前（パス名）を与える。

例。 ディレクトリ 'TestDir' の複製 'TestDir2' を作成する

```
>>> shutil.copytree( 'TestDir', 'TestDir2' ) [Enter]  ←複製処理
'TestDir2'          ←複製のディレクトリ名が返される
```

copytree の実行に先立って「複製ディレクトリ」が存在してはならない。「複製ディレクトリ」が存在している状態で copytree 関数を実行すると FileExistsError というエラーが発生する。

copytree 関数は、ディレクトリ内の個々のファイルの複製に copy2 関数を用いる。

4.26.3 書庫ファイル（アーカイブ）の取り扱いと圧縮処理に関すること

データの長期保存などの目的で、ファイルやフォルダに可逆的な圧縮処理²³⁰を施すことがある²³¹。また、複数の保存対象のファイルやフォルダを 1 つの書庫ファイル（アーカイブ）に纏めることがある。書庫ファイルの形式としては ZIP、tar といったものが有名であり、shutil はこれらの書庫の形式に対応している。

²³⁰ 伸長した際に元と同じデータが得られる圧縮処理

²³¹ ソフトウェアの配布などの場合においても、必要なファイル群が書庫ファイルの形式で扱われることが多い。

4.26.3.1 書庫（アーカイブ）の作成

make_archive 関数を使用することで、指定したディレクトリの書庫（アーカイブ）を作成することができる。

書き方： `shutil.make_archive(書庫名, root_dir=対象ディレクトリ, format=書庫形式)`

「対象ディレクトリ」の内容を纏め、「書庫名」に指定した書庫（アーカイブ）ファイルを作成する。このとき「書庫形式」に 'zip', 'tar', 'gztar', 'bztar' など²³² を指定することができる。

例. ディレクトリ 'TestDir' の ZIP アーカイブ 'TestDir.zip' を作成する

```
>>> shutil.make_archive('TestDir',root_dir='TestDir',format='zip') Enter ←アーカイブの作成
'C:\¥¥Users¥¥katsu¥¥TestDir.zip' ←作成されたアーカイブファイルのパスが返される
```

4.26.3.2 書庫（アーカイブ）の展開

unpack_archive 関数を使用することで書庫ファイルを展開することができる。

書き方： `shutil.unpack_archive(書庫ファイル名, extract_dir=展開先ディレクトリ)`

「書庫ファイル名」に指定した書庫ファイルの内容を「展開先ディレクトリ」の下に展開する。「展開先ディレクトリ」を省略すると、カレントディレクトリに展開するので注意すること。

例. 書庫ファイル 'TestDir.zip' の内容をディレクトリ 'TestDir' の下に展開する

```
>>> shutil.unpack_archive('TestDir.zip',extract_dir='TestDir') Enter ←アーカイブの展開
```

処理の結果の値は無い。(None となる)

shutil は他にも多くの機能を提供している。詳しくは Python の公式ドキュメントを参照のこと。

ZIP 形式書庫を扱うには、次に説明する zipfile モジュールを利用する方法もある。

4.27 ZIP 書庫の扱い： zipfile モジュール

ZIP 書庫を扱うための簡単な方法を提供する zipfile モジュールが Python に標準的に提供されている。このモジュールを使用するには

```
import zipfile
```

として Python 処理系に読み込む。

4.27.1 ZIP 書庫ファイルを開く

ZIP 書庫ファイルを開くには ZipFile を使用する。

書き方： `ZipFile(書庫名, モード, compression=圧縮形式)`

「書庫名」には書庫のファイル名（パス名）を指定する。「モード」にはファイルオープンの際（open 関数）のモードと同じように、

 'w'：新規作成、 'r'：既存書庫の読み込み、 'a'：既存書庫への追加

を指定する。「圧縮形式」には表 44 のようなものを指定する。

表 44: 圧縮形式			
形式	解説	形式	解説
ZIP_STORED	圧縮なし（デフォルト）	ZIP_DEFLATED	ZIP 圧縮（要 zlib モジュール）
ZIP_BZIP2	BZIP2 圧縮（要 BZ2 モジュール）	ZIP_LZMA	LZMA 圧縮（要 lzma モジュール）

※ 形式の先頭にモジュール名の接頭辞 'zipfile.' を付けること。

ZipFile を実行すると ZipFile オブジェクトが返され、以降はこれに対してファイル（メンバ）の追加や展開、読み出しなどの処理を行う。書庫に対する処理が終われば ZipFile オブジェクトに対して close メソッドを実行して閉じておく。

以下のようなサンプルデータ（テキストファイル）を書庫に加える例を挙げて使用方法を示す。

²³²'tar' はデータ圧縮をしない形式。'gztar', 'bztar' は圧縮処理にそれぞれ gzip, bzip2 を使用する。

ファイル：ziptest1.txt

```
1 First file:
2 ziptest1.txt
```

ファイル：ziptest2.txt

```
1 Second file:
2 ziptest2.txt
```

例. ZIP 書庫を新規に作成する

```
>>> import zipfile Enter ←モジュールの読み込み
>>> z = zipfile.ZipFile( 'ziptest.zip', 'w', compression=zipfile.ZIP_DEFLATED ) Enter
```

これで書庫 'ziptest.zip' が新規に作成され、その ZipFile オブジェクトが z に得られる。

4.27.2 書庫へのメンバの追加

書庫にメンバ（ファイル）を加えるには、当該書庫の ZipFile オブジェクトに対して write メソッドを実行する。

例. 書庫にファイル 'ziptest1.txt' を加える（先の例の続き）

```
>>> z.write('ziptest1.txt') Enter ←メンバの追加
```

このように write メソッドの引数に追加対象のファイル名を与える。このメソッドを次々と実行することで複数のメンバを追加することができる。

4.27.3 書庫の内容の確認

書庫の ZipFile オブジェクトに対して namelist メソッドを実行することで、書庫内のメンバ名のリストが得られる。

例. 書庫の内容の確認（先の例の続き）

```
>>> z.namelist() Enter ←メンバの確認
['ziptest1.txt'] ←メンバが1つ存在している
>>> z.close() Enter ←書庫を閉じる
```

この例では最後に書庫を close で閉じているが、再度開いて2つ目のメンバを追加する例を示す。

例. 既存の書庫を追加モードで開いてメンバを追加する（先の例の続き）

```
>>> z = zipfile.ZipFile( 'ziptest.zip', 'a' ) Enter ←既存の書庫を追加モードで開く
>>> z.write('ziptest2.txt') Enter ←メンバの追加
>>> z.namelist() Enter ←メンバの確認
['ziptest1.txt', 'ziptest2.txt'] ←メンバが2つ存在している
>>> z.close() Enter ←書庫を閉じる
```

4.27.4 書庫のメンバの読み込み

書庫のメンバをファイルのように開いてその内容を読み込むことができる。

例. 書庫のメンバの内容の読み込み（先の例の続き）

```
>>> z = zipfile.ZipFile( 'ziptest.zip', 'r' ) Enter ←書庫を読み込みモードで開く
>>> f = z.open('ziptest1.txt') Enter ←メンバをファイルのように開く
>>> b = f.read() Enter ←メンバの内容を読み込む
>>> txt = b.decode('utf-8') Enter ←エンコーディングを指定してテキストに変換
>>> print( txt ) Enter ←内容確認
First file: ←内容表示
ziptest1.txt

>>> f.close() Enter ←メンバを閉じる
>>> z.close() Enter ←書庫を閉じる
```

この例のように、書庫の ZipFile オブジェクトに対して open メソッドを実行してメンバをファイルのように開くことができる。

4.27.5 書庫の展開

書庫内の指定したメンバを展開する場合は `extract` メソッドを使用する。

書き方： `ZipFile` オブジェクト.`extract(展開対象メンバ, 展開先ディレクトリ)`

書庫内の「展開対象メンバ」を「展開先ディレクトリ」に展開する。実行後は展開先のパスを文字列で返す。

例. メンバを1つ展開する（先の例の続き）

```
>>> z = zipfile.ZipFile( 'ziptest.zip', 'r' )  Enter  ←書庫を読み込みモードで開く
>>> r = z.extract( 'ziptest1.txt', 'zipext' )  Enter  ←メンバの展開処理
>>> z.close()  Enter  ←書庫を閉じる
>>> r  Enter  ←展開先の確認
'zipext¥ziptest1.txt'  ←展開先
```

書庫の内容を全て展開する場合は `extractall` メソッドを使用する。

書き方： `ZipFile` オブジェクト.`extractall(展開先ディレクトリ)`

実行後は値を返さない。

例. 書庫の内容を全て展開する（先の例の続き）

```
>>> z = zipfile.ZipFile( 'ziptest.zip', 'r' )  Enter  ←書庫を読み込みモードで開く
>>> z.extractall( 'zipext' )  Enter  ←書庫の内容を全て展開
>>> z.close()  Enter  ←書庫を閉じる
```

4.27.5.1 パスワードで保護された ZIP 書庫へのアクセス

パスワードで保護（暗号化）された ZIP 書庫のメンバにアクセスする場合は、そのためのメソッドにキーワード引数 `'pwd=パスワード'` を与える。この場合の「パスワード」はバイト列の形で与える。例えば `'password'` というパスワードで保護された ZIP 書庫にアクセスする場合は

```
p = 'password'.encode('utf-8')  ← UTF-8 でエンコーディングされている場合
```

などとしてパスワードをバイト列に変換する。この例ではバイト列形式のパスワードを変数 `p` に得ている。これを用いて以下のような形で ZIP 書庫にアクセスする。

例. `ZipFile` オブジェクト `z`（パスワード付き）へのアクセス

- メンバ `aaa.txt` のオープン : `f = z.open('aaa.txt', pwd=p)`
- メンバ `aaa.txt` の展開 : `r = z.extract('aaa.txt', 'zipext', pwd=p)`
- 全てのメンバの展開 : `z.extractall('zipext', pwd=p)`

【補足】

階層的ディレクトリの内容を一括して ZIP 書庫にする機能（メソッド、関数など）は `zipfile` モジュールには提供されていないので、ディレクトリの内容を列举する処理を再帰的に行いながら順次メンバを ZIP 書庫に追加する手順を踏む必要がある。ディレクトリの内容をまとめて ZIP 書庫にするには、先に説明した `shutil` モジュールを使用する方が便利である。

`tar` 形式の書庫を扱う `tarfile` モジュールも標準的に提供されている。これに関しては Python の公式インターネットサイトなどの情報を参照のこと。

4.28 コマンド引数の扱い： argparse モジュール

Python でコマンドツールを作成する場合に、当該コマンドの起動の際に受け取る引数を解析する機能が必要となる。最も素朴な方法としては「2.7.9 コマンド引数の取得」(p.128) で解説した通り、sys モジュールの argv からコマンド引数の並びを取得するという方法があるが、実用的なコマンドツールを作成するには、そのコマンドの使用方法に関する情報提示機能（ヘルプ機能）を持たせたり、ハイフン付きのコマンド引数（コマンドラインオプション）の解析や、デフォルト値の設定など、様々な機能を実現する必要がある。

Python 処理系に標準的に添付されている argparse モジュールは、Python スクリプトが起動時に受け取る引数を取り扱うための各種の高度な機能を提供する。ここでは argparse モジュールの最も初歩的な使用方法について説明する。

4.28.1 コマンド引数の形式

OS のシェルから Python スクリプトを起動する際の書式（コマンドラインの書式）としては

`py スクリプト名.py コマンド引数の列` (PSF の Windows 版 Python の場合)

あるいは、

`python スクリプト名.py コマンド引数の列` (Linux, macOS²³³, Anaconda などの場合)

という形を取る。この際の「コマンド引数の列」は空白文字（スペース）で区切られた複数の文字列である。

コマンド引数は、先頭にハイフンを 1 つ、あるいは 2 つ伴う場合があり、それらを**オプション引数**²³⁴（optional arguments）と呼ぶ。オプション引数はその名の通り省略可能なものである。例えば、多くのコマンドツールが実現している情報提示機能（ヘルプ機能）として `-h`（あるいは `-help`）オプションがあり、次のようなコマンドラインの形で使用する。

`python スクリプト名.py -h`

このような形で起動すると多くのコマンドツールは使用方法などを表示して終了する。オプション引数は通常の場合、先頭にハイフン 2 つ `--` を持つ。また、その省略名は通常の場合、先頭にハイフン 1 つ `-` を持つ。

オプション引数以外のコマンド引数を**位置引数**（positional arguments）と呼ぶ。

4.28.2 使用方法

argparse モジュールは次のようにして読み込む。

```
import argparse
```

その後、**引数パーサ**（Argument Parser）オブジェクトを生成する。

《引数パーサの生成》

```
argparse.ArgumentParser(description='当該スクリプトに関する説明文')
```

説明文を与える引数は省略することができる。

例. 引数パーサ PRS を生成する

```
PRS = argparse.ArgumentParser(description='当該スクリプトに関する説明文')
```

引数パーサには初めに次のような処理を行う。

- 1) 受け付けるオプション引数や位置引数の設定： `add_argument` メソッド
- 2) コマンドラインの解析処理： `parse_args` メソッド

以上の処理は、作成するスクリプトの冒頭部分に記述しておく。

4.28.2.1 オプション引数の設定

受け付けるオプション引数を設定するには `add_argument` を使用する。

書き方： `引数パーサ.add_argument('-省略名', '--オプション名', help='説明文')`

²³³ macOS では「python3」コマンドで起動することがある。

²³⁴ コマンドラインオプションと呼ぶこともある。

「help= 説明文」は省略することができる。

例. 引数パーサ PRS にオプション '-f (省略名) / --first-option (オプション名)' を設定する
PRS.add_argument('-f', '--first-option', help=' 一番目のオプション')

4.28.2.2 位置引数の設定

受け付ける位置引数を設定するには add_argument を使用する。

書き方: 引数パーサ.add_argument('引数名', help=' 説明文')

「help= 説明文」は省略することができる。

例. 引数パーサ PRS に位置引数 'args' を設定する

```
PRS.add_argument( 'args', help=' 位置引数です. ' )
```

4.28.2.3 コマンドラインの解析処理

引数パーサに各種の引数を設定した後は、parse_args メソッドでコマンドラインを解析する。

例. 引数パーサ PRS でコマンドラインを解析する

```
args = PRS.parse_args()
```

この例では、コマンドラインの解析結果を args に得ている。parse_args メソッドの戻り値は Namespace クラスのオブジェクトであり、このオブジェクトのプロパティ（オプション引数と同名）を参照することで引数の値が得られる。例えば、先の例で設定した '--first-option' オプションの値は

```
args.first_option
```

として参照できる。

注意) オプション引数の名前の途中にハイフン '-' を含む場合は、参照時にアンダースコア '_' になる。

4.28.3 サンプルプログラムに沿った説明

次のプログラム argparse01.py はオプション引数と位置引数を受け取り、それらの値を表示するものである。

プログラム: argparse01.py

```
1 # coding: utf-8
2 import argparse      # モジュールの読み込み
3 # 初期化
4 PRS = argparse.ArgumentParser(
5     description='これは argparse モジュールのテストです. ' )
6 # 使用する引数の設定
7 PRS.add_argument( '-f', '--first-option', help=' 一番目のオプションです. ' )
8 PRS.add_argument( '-s', '--second-option', action='store_true',
9     help=' 二番目のオプションです. ' )
10 PRS.add_argument( 'a', nargs='*', help=' 位置引数の並びです. ' )
11
12 # 引数の解析
13 args = PRS.parse_args()
14
15 # 位置引数の内容確認
16 print( 'a:', args.a )
17 # オプション引数の内容確認
18 print( '--first-option:', args.first_option )
19 print( '--second-option:', args.second_option )
```

引数を与えずにこのプログラムを実行すると次のような結果となる。

実行例. py argparse01.py

```
a: []                                ←位置引数の並びは空
--first-option: None                 ←引数の値は None (引数なし)
--second-option: False               ←引数の値は False (偽)
```

位置引数は複数受け取ることができ、それらがリストとして得られる。プログラムの 10 行目で add_argument メソッドにキーワード引数 nargs='*' を与えているが、これは複数の位置引数を受け取ることができ、かつ全ての位置引数が省略可能であることを設定するものである。'nargs=' の指定方法を表 45 に示す。

表 45: add_argument メソッドのキーワード引数 'nargs='

値	解説
'*'	複数の位置引数を受け取る。(全て省略可能)
'+'	複数の位置引数を受け取る。少なくとも 1 つは引数を与える必要がある。

add_argument メソッドのキーワード引数 'nargs=' を省略すると、1 つの位置引数を受け取る形になる。

次に、サンプルプログラム argparse01.py にコマンド引数を与えて起動する例を示す。

実行例. py argparse01.py -f 1 -s 2 3

```

a: ['2', '3']           ← 2 つの位置引数が得られた
--first-option: 1       ← 引数の値は '1' (文字列)
--second-option: True   ← 引数の値は True (真)

```

オプション引数 '--first-option' は直後に 1 つの値を取る。上の実行例では '1' が得られている。これに対して '--second-option' は直後の値を取らない。(直後に続く引数は位置引数とみなされる) これはプログラムの 8 行目にあるように、add_argument メソッドにキーワード引数 action='store_true' を与えていることによるもので、この場合の引数の値は真理値 (オプション引数を与えられたときに True) となる。このようなオプション引数を特にフラグ (flag) と呼ぶ。

4.28.3.1 ヘルプ機能

argparse の引数パーサはヘルプ機能を備えており、'-h / -help' オプションがあらかじめ実装されている。

実行例. py argparse01.py -h

```

usage: argparse01.py [-h] [-f FIRST_OPTION] [-s] [a [a ...]]

これは argparse モジュールのテストです。

positional arguments:
  a                  位置引数の並びです。

optional arguments:
  -h, --help          show this help message and exit
  -f FIRST_OPTION, --first-option FIRST_OPTION
                        一番目のオプションです。
  -s, --second-option 二番目のオプションです。

```

このようなヘルプメッセージが自動的に生成される。

引数パーサオブジェクトに対して print_help メソッドを実行することでヘルプメッセージを表示することもできる。

4.28.3.2 コマンド引数の型の指定

位置引数やオプション引数に与える値はその型を限定することができる。このことに関するサンプルプログラム argparse02.py を示す。

プログラム: argparse02.py

```

1  # coding: utf-8
2  import argparse      # モジュールの読み込み
3  # 初期化
4  PRS = argparse.ArgumentParser(
5      description='これは argparse モジュールのテストです。')
6  # 使用する引数の設定
7  PRS.add_argument('-s', '--seisu', type=int,
8                  help='整数型のオプションです。', default=0)
9  PRS.add_argument('-m', '--moji', type=str,
10                  help='文字列型のオプションです。', default='x')
11 PRS.add_argument('word', type=str, help='文字列型の位置引数です。')
12 PRS.add_argument('num', type=float, help='float 型の位置引数です。')
13
14 # 引数の解析
15 args = PRS.parse_args()
16
17 # 位置引数の内容確認
18 print('word:', args.word, type(args.word))

```

```

19 print( 'num:',args.num,type(args.num) )
20 # オプション引数の内容確認
21 print( '--seisu:', args.seisu, type(args.seisu) )
22 print( '--moji:', args.moji, type(args.moji) )

```

位置引数のみ与えてこのプログラムを起動した例を示す。

実行例. py argparse02.py 単語 3.1415

```

word: 単語 <class 'str'>          ←引数の値は '単語' (文字列型)
num: 3.1415 <class 'float'>       ←引数の値は 3.1415 (float 型)
--seisu: 0 <class 'int'>         ←引数の値は 0 (整数型)
--moji: x <class 'str'>          ←引数の値は 'x' (文字列型)

```

このプログラムの中で add_argument メソッドにキーワード引数 'type=型名' を記述している。このような方法でコマンド引数に受け取る値の型を限定することができる。次の例のようにプログラムの起動時に適切でない型の値を与えるとエラーが発生し、プログラムは終了する。

実行例. py argparse02.py 単語 三. 一四

```

usage: argparse02.py [-h] [-s SEISU] [-m MOJI] word num
argparse02.py: error: argument num: invalid float value: '三. 一四'  ←型が適切でない旨のエラー

```

オプション引数は省略することができるが、その際のデフォルト値（暗黙値）を与えるには add_argument メソッドにキーワード引数 'default=値' を与える。

4.28.4 サブコマンドの実現方法

コマンドツールの中には、異なる複数の機能を持ち、使用する機能を起動時に選択するものがある。そのような類のコマンドツールは、起動時のコマンド引数にサブコマンドを与える形態（サブコマンドによって使用する機能を選択する）のことが多い。ここでは argparse モジュールでサブコマンドを実現する方法について説明する。

■ 基本的な手順

- 1) コマンドライン全体の引数パーサオブジェクトを作成する（これまで説明した通り）
- 2) 上記の引数パーサの下にサブコマンド毎に引数パーサを作成する（入れ子の引数パーサ）
- 3) サブコマンド毎の引数パーサに処理のための関数（ハンドラ）を設定する
- 4) 必要に応じてサブコマンド毎の引数パーサにオプション引数や位置引数を設定する

サブコマンドを実現する具体的な方法について argparse03.py を例に挙げて説明する。

プログラム：argparse03.py

```

1  # coding: utf-8
2  import argparse      # モジュールの読み込み
3  #--- 初期化（引数パーサの作成）---
4  PRS = argparse.ArgumentParser(
5      description='これはサブコマンドを実現するサンプルです。')
6  #--- サブコマンド用のハンドラ（関数）---
7  def cmd_sub1():
8      print(' 関数 cmd_sub1 が実行されました。')
9  def cmd_sub2():
10     print(' 関数 cmd_sub2 が実行されました。')
11
12 #--- サブコマンド毎の引数パーサの作成 ---
13 SPRS = PRS.add_subparsers()      # これに子の引数パーサを登録する
14 # サブコマンド：sub1
15 SP_sub1 = SPRS.add_parser('sub1', help='これは関数 cmd_sub1 を起動します。')
16 SP_sub1.set_defaults(handler=cmd_sub1)      # 'handler' というプロパティを設定
17 # サブコマンド：sub2
18 SP_sub2 = SPRS.add_parser('sub2', help='これは関数 cmd_sub2 を起動します。')
19 SP_sub2.set_defaults(handler=cmd_sub2)      # 'handler' というプロパティを設定
20
21 #--- 引数の解析 ---
22 args = PRS.parse_args()
23 if hasattr(args, 'handler'):
24     args.handler()      # 'handler' プロパティに登録されている関数を起動する

```

```

25 | else:
26 |     PRS.print_help()      # 未知のサブコマンドの場合はヘルプ表示

```

このプログラムはサブコマンド 'sub1', 'sub2' によって起動する関数 `cmd_sub1`, `cmd_sub2` を選択するものである。サブコマンドを受け付けるための準備として 13 行目にあるように、最上位の引数パーサに対して `add_subparsers` メソッドを実行して `ArgumentParser` オブジェクト `SPRS` を作成している。以後はこの `SPRS` に対してサブコマンド用の引数パーサオブジェクトを登録する。

プログラムの 15 行目で `add_parser` メソッドを使用してサブコマンド 'sub1' 用の引数パーサオブジェクト `SP_sub1` を生成 (`SPRS` 配下) している。また 16 行目で `SP_sub1` に対して `set_defaults` メソッドを使用して起動する関数 (ハンドラ) `cmd_sub1` を 'handler' というプロパティに登録している。サブコマンド 'sub1' の場合と同様の方法で、サブコマンド 'sub2' のための登録処理を 18,19 行目に記述している。

24 行目で `Namespace` オブジェクト `args` の 'handler' プロパティを参照してハンドラを実行している。これは 16,19 行目で設定したプロパティである。

このプログラムの実行例を示す。

例. ヘルプメッセージの表示: `py argparse03.py -h`

```
usage: argparse03.py [-h] {sub1,sub2} ...
```

これはサブコマンドを実現するサンプルです。

```
positional arguments:
```

```
{sub1,sub2}
```

```
sub1      これは関数 cmd_sub1 を起動します.
```

```
sub2      これは関数 cmd_sub2 を起動します.
```

```
optional arguments:
```

```
-h, --help show this help message and exit
```

例. サブコマンド 'sub1' の指定: `py argparse03.py sub1`

関数 `cmd_sub1` が実行されました。

例. サブコマンド 'sub2' の指定: `py argparse03.py sub2`

関数 `cmd_sub2` が実行されました。

例. 登録されていないサブコマンド 'sub3' の指定: `py argparse03.py sub3`

```
usage: argparse03.py [-h] {sub1,sub2} ...
```

```
argparse03.py: error: argument {sub1,sub2}: invalid choice: 'sub3' (choose from 'sub1', 'sub2')
```

ここで紹介した機能以外にも多くの機能を `argparse` モジュールは提供している。詳しくは Python の公式インターネットサイトなどを参照のこと。

4.29 スクリプトの終了 (プログラムの終了)

4.29.1 sys.exit 関数

Python スクリプト (プログラム) の実行を終了するには `sys` モジュールの `exit` を使用する。Python スクリプト中でこの関数を実行した時点で当該スクリプトの実行は終了する。

書き方: `sys.exit(終了コード)`

「終了コード」²³⁵ は、当該スクリプトを起動したシェル環境 (コマンドウィンドウ, ターミナルウィンドウ) に通知される。

例. Windows のコマンドプロンプトウィンドウでの終了コードの確認

```
echo %ERRORLEVEL%
```

例. bash (Linux, macOS) での終了コードの確認

```
echo $?
```

²³⁵ 終了ステータスとも言う。多くの場合、プログラムが正常終了する際は終了コードは 0 とする。

sys.exit は SystemExit 例外を起こしてプログラムを強制的に終了させる。

4.29.2 os._exit 関数

先の sys.exit とは別に、Python のプロセスを強制的に終了させるものに os._exit 関数 (os モジュールの _exit 関数) がある。Python スクリプト中でこの関数を実行した時点で当該スクリプトの実行は終了するが、SystemExit 例外は発生しない。

書き方： `os._exit(終了コード)`

4.29.3 スクリプト終了時に実行する処理： atexit モジュール

atexit モジュールを利用すると、Python スクリプトが終了する時点で実行する処理を登録することができる。このモジュールは利用に先立って

```
import atexit
```

として読み込んでおく。スクリプト終了時に実行する処理は register 関数を用いて登録する。

書き方： `register(関数名, 引数並び…)`

スクリプト終了時に実行する処理は関数として定義しておき、その関数名を第 1 引数の「関数名」に与え、その関数に与える引数を続けて書き並べる。「引数並び」にはキーワード引数を与えることもできる。

register 関数は複数実行することもでき、複数の処理を登録することができる。ただしその場合は、register で登録した順序とは逆の順序で、スクリプト終了時に処理が実行される。

register 関数の使用例をサンプルプログラム atexit01.py に示す。

プログラム： atexit01.py

```
1  # coding: utf-8
2  import sys
3  import os
4  import atexit
5
6  #--- 終了処理のための関数 ---
7  def exFunc( x, y, kw='デフォルトメッセージ' ):
8      print( '終了処理: '+x+'> '+y, ',kw='+kw )
9
10 #--- 終了処理の登録 ---
11 atexit.register( exFunc, 'A', '1番目に登録した処理' )
12 atexit.register( exFunc, 'B', '2番目に登録した処理', kw='2番目に実行' )
13 atexit.register( exFunc, 'C', '3番目に登録した処理', kw='1番目に実行' )
14
15 #--- 主プログラム ---
16 print('主プログラムからの出力')
17 print('-----')
18
19 #sys.exit(1)
20 #os._exit(1)
```

このプログラムの実行例を次に示す。

実行例.

主プログラムからの出力

終了処理:C> 3番目に登録した処理 ,kw=1番目に実行

終了処理:B> 2番目に登録した処理 ,kw=2番目に実行

終了処理:A> 1番目に登録した処理 ,kw=デフォルトメッセージ

注意)

atexit.register で登録された処理（終了時処理）は、Python スクリプトが正常に終了する際に実行される。すなわち、sys.exit で終了する際にも終了時処理は実行されるが、os._exit で終了する際には実行されない。これらのことを、先のプログラム atexit01.py の 19~20 行目のコメント記号「#」を交互に外して確認されたい。

4.30 Python の型システム

4.30.1 type 型オブジェクト

Python のデータ型は `type` 関数で調べることができる。

例. 整数値 1 の型を調べる

```
>>> t = type( 1 )  Enter  ← type 関数による型の調査
>>> t              Enter  ←内容確認
<class 'int'>      ←型
```

整数値の 1 の型は `int` であり、この型を意味するオブジェクト `<class 'int'>` が得られている。

ところで、この `<class 'int'>` もまた Python のオブジェクト (**type 型のオブジェクト**) であり、これの型を同様の方法で調べることができる。(次の例)

例. `<class 'int'>` の型を調べる (先の例の続き)

```
>>> t2 = type( t )  Enter  ← type 関数による型の調査
>>> t2              Enter  ←内容確認
<class 'type'>      ←型
```

型を意味するオブジェクトの型は `<class 'type'>` であることがわかる。

Python では型名を直接参照することもできる。

例. 型名そのものを参照 (先の例の続き)

```
>>> int             Enter  ←「int」を参照
<class 'int'>       ←型を表すオブジェクト
>>> float           Enter  ←「float」を参照
<class 'float'>     ←型を表すオブジェクト
```

型を表すオブジェクト `<class ...>` から型名を表す文字列を得るには `__name__` プロパティを参照する。

例. 型名の取り出し (先の例の続き)

```
>>> t.__name__      Enter  ←型名の取り出し
'int'               ←型名
>>> t2.__name__     Enter  ←型名の取り出し
'type'              ←型名
```

`type` 型のオブジェクトを、それが表すクラスのコンストラクタとして使用できる。

例. タプルを表す `type` 型オブジェクトをコンストラクタとして使う

```
>>> t = type( (1,2) ) Enter  ←タプルの型を取得
>>> t              Enter  ←内容確認
<class 'tuple'>    ←タプル
>>> t( ['a','b','c'] ) Enter  ←それを用いてタプルを生成
('a', 'b', 'c')    ←得られたタプル
```

4.30.2 型の階層 (クラス階層)

オブジェクト指向の枠組みでは、全ての型は階層的に分類されている。先の「2.9 オブジェクト指向プログラミング」(p.154)でも解説したように、クラスには継承関係 (スーパークラスとそれに属するサブクラスの関係) があるが、プログラマが作成したクラス定義のみならず、数値や文字列を始めとする組み込みのデータ型も全て1つのクラス階層の中に位置づけられている。

Python では全てのデータ型 (クラス) は `object` クラスの配下に位置づけられる。このことは、クラスの継承関係を調べる `issubclass` 関数で確認することができる。

例. object クラス配下にクラスが位置づけられていることの確認

```
>>> issubclass(int,object) Enter ← int クラスが object クラス配下にあるか確認
True                               ←結果 (object クラス配下にある)

>>> class C: Enter ←クラス C の定義
...     pass Enter ←定義 (便宜上 pass とする)
... Enter ←クラス定義の終了

>>> issubclass(C,object) Enter ←クラス C が object クラス配下にあるか確認
True                               ←結果 (object クラス配下にある)
```

4.30.2.1 スーパークラス, サブクラスを調べる方法

あるクラスのスーパークラスを調べるには, そのクラスの `__bases__` プロパティを参照する. またサブクラスを調べるには, そのクラスに対して `__subclasses__` メソッドを実行する. 次の例は, `int` 型のスーパークラスとサブクラスを調べるものである.

例. `int` 型のスーパークラスとサブクラスの調査

```
>>> int.__bases__ Enter ← int のスーパークラスを参照
(<class 'object'>,) ←スーパークラスのタプル

>>> int.__subclasses__() Enter ← int のサブクラスを調べる
[<class 'bool'>, <enum 'IntEnum'>, <enum 'IntFlag'>, ←サブクラスのリスト
 <class 'sre_constants.NamedIntConstant'>, <class 'subprocess.Handle'>]
```

このように, `__bases__` プロパティからスーパークラスのタプルが得られ, `__subclasses__` メソッドの実行結果からサブクラスのリストが得られる.

注意) 実行時の Python 処理系の状態によって出力結果が異なることがある.

【参考】

`__subclasses__` メソッドは対象とするクラスによって仕様が異なることがある. 例えば `type` クラスに対する `__subclasses__` メソッドの実行結果は次のようになる.

例. `type` クラスに対する `__subclasses__` メソッド

```
>>> type.__subclasses__() Enter ←空の引数で実行を試みると…
Traceback (most recent call last): ←引数が不足している旨のエラー
  File "<stdin>", line 1, in <module>   が発生する
TypeError: unbound method type.__subclasses__() needs an argument

>>> type.__subclasses__(int) Enter ←引数「int」を与えて実行
[<class 'bool'>, <enum 'IntEnum'>, <enum 'IntFlag'>, ←int のサブクラスのリスト
 <class 'sre_constants.NamedIntConstant'>, <class 'subprocess.Handle'>]
```

このように

```
type.__subclasses__(クラス)
```

と, 引数にクラスを与えると, そのクラスの配下のサブクラスのリストが得られる. これを応用すると, クラス階層を再帰的に調べることができる. そのためのサンプルプログラム `getSubclasses.py` を次に示す.

プログラム: `getSubclasses.py`

```
1 # coding: utf-8
2 def k(o): return o.__name__      # クラス名の文字列を取得する関数
3
4 # クラス階層の出力
5 def getSubclasses(cls, n=0):
6     sc = sorted(type.__subclasses__(cls), key=k)
7     for scls in sc:
8         print(' '*n+str(scls))
9         getSubclasses(scls, n+1)
10
11 #
12 if __name__ == '__main__':
13     getSubclasses(object)
```


このプログラムをスクリプトとして実行すると、object クラス配下の全てのサブクラスが階層的に出力される。また、Python 処理系にモジュールとして読み込み、getSubclasses 関数の引数にクラスを与えると、そのクラスの配下のサブクラスが階層的に出力される。

例. int 配下のサブクラスを表示 (getSubclasses.py はカレントディレクトリにあるものとする)

```
>>> from getSubclasses import getSubclasses Enter    ←モジュールとして読み込む
>>> getSubclasses(int) Enter    ← int 配下のサブクラスを出力
<enum 'IntEnum'>
  <enum '_ParameterKind'>
  <enum '_Precedence'>
<flag 'IntFlag'>
  <flag 'BufferFlags'>
  <flag 'RegexFlag'>
<class 're._constants._NamedIntConstant'>
<class 'bool'>
```

注意. Python の版や実行状態によってクラス階層が異なることがある。

課題.

先の例のように Python の対話モードで getSubclasses(object) として Python 処理系で定義されている全てのクラス階層を表示して眺めよ。

4.30.3 数のクラス階層： numbers モジュール

Python 言語で扱える数値の型には int, float, complex などがある。それらのクラスは object クラス直下のものであり、Python の数値のクラスの階層は数学的な分類に沿ったものではない。しかし、標準ライブラリである numbers モジュールを使用すると、数学的な分類に沿った形で Python で扱う数値の型を判定することができる。

このモジュールは

```
import numbers
```

として読み込む。その後、numbers.Number クラスから派生する形で数のクラスが扱える。このことを、先に示したプログラム getSubclasses.py を用いて確認する。(次の例)

例. numbers.Number 以下のクラス階層

```
>>> import numbers Enter    ← numbers モジュールの読み込み
>>> from getSubclasses import getSubclasses Enter    ← getSubclasses.py の読み込み
>>> getSubclasses( numbers.Number ) Enter    ← 数の階層を閲覧
<class 'numbers.Complex'>          ← Complex (複素数)
  <class 'numbers.Real'>          ← Real (実数)
    <class 'numbers.Rational'>    ← Rational (有理数)
      <class 'numbers.Integral'>  ← Integral (整数)
```

このように、numbers.Number 配下に、numbers.Complex (複素数)、numbers.Real (実数)、numbers.Rational (有理数)、numbers.Integral (整数) が順に派生クラスになっている。

分数を扱うための標準モジュール fractions を読み込むと、分数のクラス fractions.Fraction もこの階層に組み込まれる。(確認されたい)

以下に、numbers.Number の体系に基づいて各種の数値の型を判定する例を示す。

例. 各種数値の型の判定 (先の例の続き)

```
>>> isinstance( 3, numbers.Integral ) Enter ← 3 は整数か?
True ← 真
>>> isinstance( 3.14, numbers.Integral ) Enter ← 3.14 は整数か?
False ← 偽
>>> isinstance( 3.14, numbers.Real ) Enter ← 3.14 は実数か?
True ← 真
>>> isinstance( 1+3j, numbers.Real ) Enter ← 1+3j は実数か?
False ← 偽
>>> isinstance( 1+3j, numbers.Complex ) Enter ← 1+3j は複素数か?
True ← 真
```

例. fractions.Fraction (分数) の型の判定 (先の例の続き)

```
>>> from fractions import Fraction Enter ← 分数用モジュールの読み込み
>>> x = Fraction(1,3) Enter ← 分数 1/3 の生成
>>> print(x) Enter ← 確認
1/3 ← 分数が得られた
>>> isinstance( x, numbers.Integral ) Enter ← 整数か?
False ← 偽
>>> isinstance( x, numbers.Rational ) Enter ← 有理数か?
True ← 真
```

例. すべての数値は numbers.Number (数) に属することを検証する (先の例の続き)

```
>>> isinstance( x, numbers.Number ) Enter ← 分数は数か?
True ← 真
>>> isinstance( 1+3j, numbers.Number ) Enter ← 1+3j は数か?
True ← 真
>>> isinstance( 3.14, numbers.Number ) Enter ← 3.14 は数か?
True ← 真
>>> isinstance( 3, numbers.Number ) Enter ← 3 は数か?
True ← 真
>>> isinstance( '2.71828', numbers.Number ) Enter ← '2.71828' は数か?
False ← 偽
```

多倍長精度の浮動小数点数を扱うためのライブラリとして mpmath (p.25「2.4.6.13 多倍長精度の浮動小数点数の扱い」参照のこと) があるが、このライブラリが提供する浮動小数点数のクラスは、実数が mpf、複素数は mpc である。これらクラスは numbers.Number のクラス階層に統合されている。

例. mpmath の実数と複素数 (先の例の続き)

```
>>> from mpmath import mp Enter ← mpmath の読み込み
>>> mp.dps = 30 Enter ← 計算精度の設定 (30 桁)
>>> x = mp.sqrt( 2 ) Enter ←  $\sqrt{2}$  の計算
>>> print( x ) Enter ← 出力
1.41421356237309504880168872421 ←  $\sqrt{2}$  の近似値
>>> type( x ) Enter ← データ型は
<class 'mpmath.ctx_mp_python.mpf'> ← mpf である
>>> c = mp.mpc( 1/x, 1/x ) Enter ←  $1/\sqrt{2} + (1/\sqrt{2})i$  の計算
>>> print( c ) Enter ← 出力
(0.707106781186547524400844362105 + 0.707106781186547524400844362105j) ← 複素数
>>> type( c ) Enter ← データ型は
<class 'mpmath.ctx_mp_python.mpc'> ← mpc である
```

これらクラスが numbers.Number の派生クラスとなっていることを確認する。(次の例)

例. numbers の機能による mpf, mpc の判定 (先の例の続き)

```
>>> isinstance( x, numbers.Integral )  ← mpf は整数か？
False                                ← 偽
>>> isinstance( x, numbers.Real )  ← mpf は数か？
True                                ← 真
>>> isinstance( c, numbers.Real )  ← mpc は実数か？
False                                ← 偽
>>> isinstance( c, numbers.Complex )  ← 1+3j は複素数か？
True                                ← 真
```

高速数値演算のためのライブラリ NumPy ²³⁶ の各種の数値型も numbers.Number のクラス階層に統合されている。

例. NumPy の符号付き 64 ビット整数 int64

```
>>> import numbers  ← numbers の読み込み
>>> import numpy as np  ← NumPy の読み込み
>>> z = np.int64( 2 )  ← 符号付き 64 ビット整数の生成
>>> z  ← 確認
np.int64(2)                        ← NumPy の整数
>>> type( z )  ← 型を調べる
<class 'numpy.int64'>             ← int64 の整数
>>> isinstance( z, numbers.Integral )  ← z は整数か？
True                                ← 真
```

例. NumPy の 64 ビット浮動小数点数 float64 (先の例の続き)

```
>>> x = np.float64( 1.23 )  ← 64 ビット浮動小数点数の生成
>>> x  ← 確認
np.float64(1.23)                  ← NumPy の浮動小数点数
>>> type( x )  ← 型を調べる
<class 'numpy.float64'>          ← float64 の浮動小数点数
>>> isinstance( x, numbers.Integral )  ← x は整数か？
False                                ← 偽
>>> isinstance( x, numbers.Real )  ← x は実数か？
True                                ← 真
```

例. NumPy の 128 ビット複素数 complex128 (先の例の続き)

```
>>> c = np.complex128( 1+1j )  ← 128 ビット複素数の生成
>>> c  ← 確認
np.complex128(1+1j)               ← NumPy の複素数
>>> type( c )  ← 型を調べる
<class 'numpy.complex128'>       ← complex128 の複素数
>>> isinstance( c, numbers.Real )  ← c は実数か？
False                                ← 偽
>>> isinstance( c, numbers.Complex )  ← c は複素数か？
True                                ← 真
```

²³⁶ 拙書「Python3 ライブラリブック」でも解説しています。

5 TCP/IP による通信

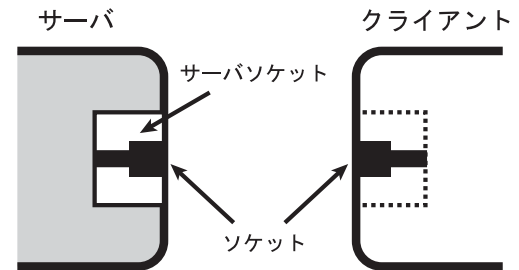
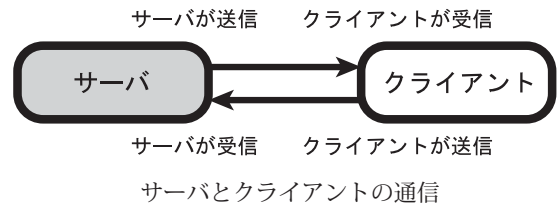
ここでは、TCP/IP による通信機能を提供するいくつかのライブラリを紹介して、それらの基本的な使用方法について説明する。

5.1 socket モジュール

TCP/IP 通信はサーバとクライアントの 2 者間の通信（右図）を基本とする。

サーバ、クライアントはそれぞれソケットを用意し、それを介して通信する。（右下の図）

サーバはクライアントからの接続要求を受け付ける（右図の「サーバソケット」がクライアントからの接続要求を受け付ける）システムであり、通常は接続待ちの状態で待機している。それに対してクライアントからの接続が要求され、接続処理が完了すると、右図の「ソケット」を介して両者間で双方向の通信が可能となる。（図 39）



サーバ、クライアントそれぞれにソケットを用意する

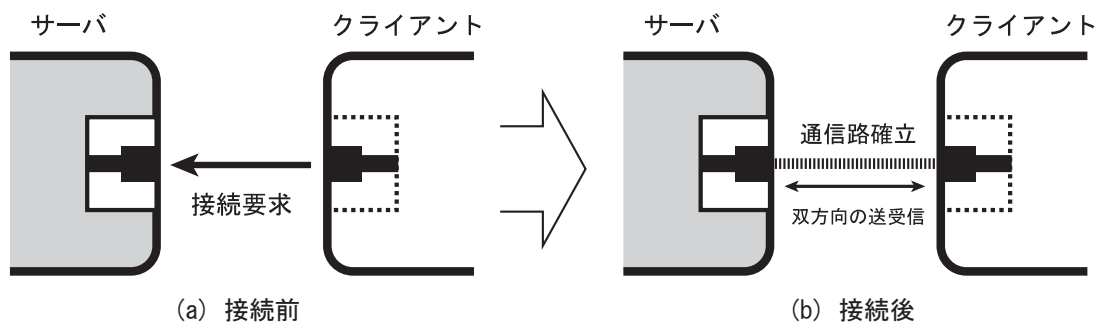


図 39: ソケットを介した接続

このような通信を実現するために Python には socket モジュールが用意されている。socket モジュールは次のようにして読み込む。

```
import socket
```

5.1.1 ソケットの用意

ソケットは socket コンストラクタで生成する。

書き方： `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

この処理が正常に終了すると、ソケットオブジェクトが生成されて返される。ソケットを用意する方法はサーバ、クライアントの両方において同じである。

5.1.2 サーバ側プログラムの処理

サーバ側ソケットにはソケットオプション²³⁷を設定する。これには次のように `setsockopt` メソッドを用いる。

書き方： `ソケットオブジェクト.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`

次に、ソケットを IP アドレスとポートにバインドする。これには次のようにして `bind` メソッドを用いる。

書き方： `ソケットオブジェクト.bind((ホストアドレス, ポート))`

「ホストアドレス」には当該計算機環境の NIC の IP アドレスを文字列で与える。この部分に '0.0.0.0' を与えると、当該計算機環境の全ての NIC が対象となる。

²³⁷ 詳しい説明は TCP/IP の関連書籍や技術資料に譲る。

ここまででサーバ側のソケット (p.367 の図の「サーバソケット」) の準備がほぼ終わる。後は `listen` メソッドを使用してクライアントからの受信の準備をする。(下記参照)

書き方： `ソケットオブジェクト.listen()`

`listen` メソッドの引数にはバックログの値を与えることができる。これは、サーバのソケットが保持できる接続の待ち行列の長さで、デフォルトで `socket.SOMAXCONN` の値が与えられている。

実際にクライアントからの接続要求を受信するには、次のように `accept` メソッドを使用する。

書き方： `ソケットオブジェクト.accept()`

この処理が完了すると、相手システム (クライアント) と通信するための新たなソケットオブジェクト (p.367 の図の「ソケット」) とアドレスのタプルが返される。ここで言うアドレスとは、クライアント側の IP アドレスとポート番号のタプルである。

サーバ側プログラムは、複数のクライアントからの接続要求を受け付けることができ、異なるクライアントから `accept` する度に、それぞれに対応するソケットオブジェクトが生成される。

5.1.3 クライアント側プログラムの処理

クライアント側では、生成したソケットに対して次のように `connect` メソッドを用いてサーバに接続を要求する。(その前にタイムアウトを設定しておく方が良い)

書き方： `ソケットオブジェクト.settimeout(秒数)`

書き方： `ソケットオブジェクト.connect((ホストアドレス, ポート番号))`

`connect` メソッドの引数には、サーバの「ホストアドレス」と「ポート番号」のタプルを与える。「ホストアドレス」には IP アドレスもしくは FQDN を文字列で与える。「ポート番号」は整数値で与える。

5.1.4 送信と受信

ソケットを介して相手システムにメッセージを送信するには `send` メソッドを使用する。(下記参照)

書き方： `ソケットオブジェクト.send(データ)`

送信する「データ」はバイトデータ (bytes 型) で与える。相手システムからのメッセージを受信するには、`recv` メソッドを使用する。(下記参照)

書き方： `ソケットオブジェクト.recv(バッファサイズ238)`

`recv` メソッドの処理が正常に終わると、受信したメッセージがバイトデータ (bytes 型) として返される。

通信が終了すると、次のように `close` メソッドを使用してソケットを閉じておく。

書き方： `ソケットオブジェクト.close()`

5.1.5 サンプルプログラム：最も基本的な通信

ここでは、サーバプログラムとクライアントプログラムが相互に接続してメッセージを交換するプログラムを示す。

サーバプログラム：test13-sv.py

```
1 # coding: utf-8
2 import socket
3
4 # 通信先 (サーバ) の情報
5 host = '127.0.0.1'
6 port = 8001
7
8 # サービスの準備
9 sSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 sSock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
11 sSock.bind((host, port))
12 sSock.listen()
13
14 # 接続の受け付け
```

²³⁸通常は 4096 にする。

```

15 print('Waiting for connection...')
16 (cSock,cAddr) = sSock.accept()
17 print('Connection accepted!: ',cAddr)
18
19 # クライアントから受信
20 r = cSock.recv(4096)
21 print( 'クライアントから> ',r.decode('utf-8') )
22
23 # クライアントへ送信
24 msg = 'はい、届いていますよ.'.encode('utf-8')
25 cSock.send(msg)
26
27 # ソケットを終了する
28 cSock.close()
29 sSock.close()

```

クライアントプログラム：test13-cl.py

```

1  # coding: utf-8
2  import socket
3
4  # 通信先（サーバ）の情報
5  host = '127.0.0.1'
6  port = 8001
7
8  # 接続処理
9  cSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 cSock.settimeout(3.0)
11 print('Connecting...')
12 cSock.connect((host, port))
13 print('Connection accepted!')
14
15 # サーバへ送信
16 msg = '受信できていますか?'.encode('utf-8')
17 cSock.send(msg)
18
19 # サーバから受信
20 r = cSock.recv(4096)
21 print( 'サーバから> ',r.decode('utf-8') )
22
23 # ソケットを終了する
24 cSock.close()

```

先にサーバプログラムを起動しておき、次にクライアントプログラムを起動すると両者が接続される。接続が確立されると、クライアント側からサーバ側に対して

「受信できていますか？」

と送信される。サーバ側はこれを受信して標準出力に出力した後、クライアント側に対して

「はい、届いていますよ。」

と送信する。クライアント側はこれを受信して標準出力に出力する。実行例を次に示す。

サーバ側の実行例：

```

Waiting for connection...
Connection accepted!: ('127.0.0.1', 55735)
クライアントから> 受信できていますか？

```

クライアント側の実行例：

```

Connecting...
Connection accepted!
サーバから> はい、届いていますよ。

```

ここに示したサンプルは、クライアント側が一度だけメッセージをサーバに送信し、それを受けたサーバ側が一度だけクライアントにメッセージを送信して終了するものである。実用的な通信においては、クライアントとサーバがデータのやり取りを複数回繰り返すだけでなく、1つのサーバが複数のクライアントからの接続要求を受け付け、各クライアントとの通信を個別に実行する。次に、実用的な通信処理のための具体的な方法について解説する。

5.1.6 実用的な通信機能を実現する方法

実用的なサーバクライアント通信においては、1つのサーバが複数のクライアントからの接続要求を同時に受け付け、個々のクライアントを識別して通信を行う。また、通信処理には待ち時間が発生することが一般的である。このように、待ち時間が発生する複数の処理を並行して実行するには非同期処理の形態で通信システムを構築するのが一般的である。

5.1.6.1 socket ライブラリと asyncio ライブラリの併用

ここでは、先の「4.5 非同期処理：asyncio」(p.275)で解説した非同期処理を応用して通信システムを構築する方法について解説する。

■ サーバがクライアントからの接続要求を受け付ける処理

socket ライブラリでは、サーバのソケットオブジェクトに対して `accept` メソッドを実行することでクライアントからの接続要求を受け付ける。これと同様の処理を `asyncio` のイベントループ上で非同期に行うことができる。具体的には、イベントループオブジェクトに対して `sock_accept` メソッドを実行する。

書き方： `await イベントループオブジェクト.sock_accept(サーバソケット)`

「サーバソケット」による接続要求の受け付けを「イベントループオブジェクト」で非同期に実行する。接続要求を受け付けるとクライアントのソケットとアドレスのタプルを返す。この場合のアドレスは、クライアント側の IP アドレスとポート番号のタプルを意味する。戻り値に関しては socket ライブラリの `accept` メソッド (p.368) に準じている。

■ 相手が送信したデータを受信する処理

socket ライブラリでは、クライアントのソケットオブジェクトに対して `recv` メソッドを実行することで、相手が送信したデータを受信する。これと同様の処理を `asyncio` のイベントループ上で非同期に行うことができる。具体的には、イベントループオブジェクトに対して `sock_recv` メソッドを実行する。

書き方： `await イベントループオブジェクト.sock_recv(クライアントソケット, バッファサイズ)`

「クライアントソケット」からのデータの受信を「イベントループオブジェクト」で非同期に実行する。このとき受信の「バッファサイズ」を整数値で指定する。受信したデータはバイト列 (bytes 型) で返される。

相手システムが「バッファサイズ」以下の大きさのデータを送信した場合は、`sock_recv` メソッドの結果はそのデータ全体となり、「バッファサイズ」よりも大きなデータを送信した場合は、最初の「バッファサイズ」の部分だけを返し、残りは受信バッファに残される。受信バッファに残されたデータは次の `sock_recv` メソッドの実行時に読み取られる。受信バッファの残部が更に「バッファサイズ」より大きい場合は、次の `sock_recv` メソッドの実行において、残部のデータの先頭の「バッファサイズ」の大きさの部分只得られ、その残部が受信バッファに残される。(以下同様)

相手システムが接続を終了すると、自システム側の `sock_recv` メソッドは空のバイト列 `b''` を返す。これを応用することで、相手システムの終了を検知することができる。

■ 相手にデータを送信する処理

socket ライブラリでは、クライアントのソケットオブジェクトに対して `send` メソッドを実行することで、相手にデータを送信する。これと同様の処理を `asyncio` のイベントループ上で非同期に行うことができる。具体的には、イベントループオブジェクトに対して `sock_sendall` メソッドを実行する。

書き方： `await イベントループオブジェクト.sock_sendall(クライアントソケット, データ)`

「クライアントソケット」を通じて相手に「データ」を送信する処理を、「イベントループオブジェクト」で非同期に実行する。「データ」はバイト列 (bytes 型) で与える。

クライアントとサーバは互いに `recv`, `sendall` メソッドを用いて送受信を繰り返す。一連の通信を終了するにはクライアントソケットに対して `close` メソッドを実行する。

書き方： `クライアントソケット.close()`

■ クライアントがサーバに対して接続要求する処理

socket ライブラリでは、クライアントのソケットオブジェクトに対して `connect` メソッドを実行することで、サーバに対して接続要求をする。これと同様の処理を `asyncio` のイベントループ上で非同期に行うことができる。具体的

には、イベントループオブジェクトに対して `sock.connect` メソッドを実行する。

書き方： `await イベントループオブジェクト.sock.connect(クライアントソケット, (IP アドレス, ポート番号))`

クライアントが「クライアントソケット」を用いて、「IP アドレス」、「ポート番号」で指定したサーバに接続要求する。またこの処理は「イベントループオブジェクト」で非同期に実行される。

「IP アドレス」は文字列として記述する。またこの部分にサーバの FQDN を記述しても良い。

【サンプルプログラム】

socket ライブラリ, asyncio ライブラリを併用して通信を実現した例を `asyncSocketSv01.py` (サーバ), `asyncSocketCl01.py` (クライアント) に示す。

サーバプログラム： `asyncSocketSv01.py`

```
1  # coding: utf-8
2  import asyncio
3  import socket
4
5  # 個別のクライアントとの通信
6  async def handleCl( cSoc, addr ):
7      loop = asyncio.get_running_loop()          # 現行のイベントループを取得
8      while True:
9          req = await loop.sock_recv( cSoc, 1024 )    # クライアントから受信
10         msg = req.decode('utf-8')
11         print(f'{addr}: {msg}')
12         r = b'[echo] '+ req
13         await loop.sock_sendall( cSoc, r )    # クライアントへ送信
14         if msg == 'end': break                # クライアントからの終了指示を受けた場合
15         cSoc.shutdown(socket.SHUT_RDWR)        # ソケットの終了
16         cSoc.close()                          # ソケットを閉じる
17
18 # サーバの作成と起動
19 async def sv():
20     # サーバを作成し、初期設定して起動
21     sSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22     sSoc.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
23     sSoc.bind(('0.0.0.0', 8000))
24     sSoc.listen(5)
25     sSoc.setblocking(False)                  # 今回は必須ではない
26     # 複数のクライアントからの接続要求に応える処理
27     loop = asyncio.get_running_loop()        # 現行のイベントループを取得
28     while True:
29         cSoc, addr = await loop.sock_accept( sSoc )    # 接続要求の受け入れ
30         print(f'Connection from {addr}')
31         loop.create_task(handleCl( cSoc, addr ))    # 当該クライアントとの通信処理
32
33 asyncio.run( sv() ) # サーバのタスクをイベントループに投入して実行
```

クライアントプログラム： `asyncSocketCl01.py`

```
1  # coding: utf-8
2  import asyncio
3  import socket
4
5  # クライアント用コルーチン
6  async def client():
7      loop = asyncio.get_running_loop()    # 現行のイベントループの取得
8      cSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # ソケットの作成
9      await loop.sock_connect( cSoc, ('127.0.0.1', 8000) )    # サーバに接続要求
10     while True:
11         # メッセージを入力して送信する処理
12         msg = input('終了:end> ').rstrip()
13         await loop.sock_sendall( cSoc, msg.encode('utf-8') ) # サーバにデータ送信
14         res = await loop.sock_recv( cSoc, 1024 ) # 直後にサーバからのデータを受信
15         print(f'Received: {res.decode('utf-8')}')
16         if msg == 'end': break                # 入力されたデータが end ならwhileを終了
17         cSoc.shutdown(socket.SHUT_RDWR)        # ソケットを終了
18         cSoc.close()                          # ソケットを閉じる
19
20 asyncio.run( client() )    # クライアントのタスクをイベントループに投入して実行
```

クライアントプログラム `asyncSocketCl01.py` では、同じ計算機で実行されてるサーバに対して

```
await loop.sock_connect( cSoc, ('127.0.0.1', 8000) )
```

で接続した後は、コンソールから入力された文字列をバイト列に変換してサーバに送信する処理を繰り返す。このとき、'end' を入力するとクライアントプログラムが終了する。また、それを受けたサーバは接続を終了する。

サーバプログラム `asyncSocketSv01.py` では、コルーチン `sv()` 内でサーバソケットを生成してクライアントからの接続要求を受け付ける。この処理は `while` ループで繰り返され、複数のクライアントからの接続要求を次々と受け付ける。各クライアントから要求に応えるのはコルーチン `handleCl` で、この中の `while` ループによってクライアントから次々と送られてくるデータを処理している。この場合の処理は、受信したデータをクライアントにエコーバックするというものである。また、'end' を受信するとクライアントソケットを閉じて `handleCl` のタスクを終了する。`handleCl` のタスクは複数のクライアントからの要求に応じて個別に起動し、イベントループ上で非同期に実行される。

ここに示したサンプルでは、サーバ、クライアント共に、クライアントソケット `cSoc` を閉じる前に

```
cSoc.shutdown(socket.SHUT_RDWR)
```

として終了させており、より安全な形で接続を閉じている。

上のプログラムを実行するには、先にサーバプログラムを起動し、次にクライアントプログラムを起動する。サーバプログラムの起動直後はコンソールには何も出力されないが、クライアントが同じ計算機の別のコンソールで起動した際に

```
Connection from ('127.0.0.1', 57961) ←クライアント側のポート番号は実行時に適宜設定される
```

などと出力される。クライアント側のコンソールには「終了:end>」というプロンプトが表示され、ユーザに対して文字の入力を促す。これに対して、例えば「こんにちは。」と入力して「Enter」を押すとそれがサーバに送信され、サーバ側のコンソールに

```
('127.0.0.1', 57961): こんにちは。
```

などと出力される。また直後に、この内容がクライアントにエコーバックされ、クライアント側のコンソールに

```
Received: [echo] こんにちは。
```

と出力される。このような入力→送信→エコーバックのサイクルはクライアント側で 'end' と入力するまで繰り返される。

上のサーバプログラムを終了するには、コンソールから `CTRL+C` とキーボード操作を行う。今回はサーバ側に例外処理を実装していないので、サーバ終了時にエラーメッセージが表示される。

上のクライアントプログラムは、サーバと同じ計算機環境で複数起動することが可能で、各クライアントからの通信を1つのサーバが同時に対応する。(試されたい)

5.1.6.2 asyncio ライブラリのみで通信を実現する方法

`asyncio` ライブラリはその内部で `socket` ライブラリを使用して TCP/IP 通信のための機能を実現している。また `asyncio` では、イベントループ上で通信の処理を完結させることができるように API が設計されており、`socket` ライブラリを明に意識せずに通信処理を実現することができる。

■ サーバオブジェクトの生成

`asyncio` ライブラリでは独自にサーバクラス (Server クラス) を実装しており、このクラスのインスタンスを用いてサーバを実現することができる。サーバのインスタンスは `start_server` 関数 (コルーチン) で作成する。

書き方： `await asyncio.start_server(ハンドラ関数, IP アドレス, ポート番号)`

当該計算機で通信を受け付ける NIC の「IP アドレス」を文字列で指定し、「ポート番号」を整数値で指定する。「IP アドレス」に '0.0.0.0' を指定すると、当該計算機の全ての NIC が対象となる。

クライアントとの具体的な通信処理は、指定した「ハンドラ関数」(後述)で行う。`start_server` 関数は生成したサーバのインスタンス (Server オブジェクト) を返す。

■ サーバオブジェクトの起動と停止

Server オブジェクトに対して `serve_forever` メソッドを非同期に実行することで、イベントループ上でサーバが起動する。

書き方： `await Server オブジェクト.serve_forever()`

また、`close` メソッドでそれを停止することができる。ただし `close` メソッドは同期処理であり、その処理の終了を `wait_closed` メソッドで非同期に待機すること。

書き方： `Server オブジェクト.close()`
`await Server オブジェクト.wait_closed()`

■ ハンドラ関数

接続要求を受け付けたクライアントとの具体的な通信処理は、`start_server` 関数に指定したハンドラ関数で行う。この関数は 2 つの引数を取るコルーチンとして定義する。

書き方： `async 関数名 (リーダ, ライタ):`
(応答処理)

この関数はクライアントからの接続要求に対して呼び出され、その際に自動的に引数が渡される。「リーダー」は `StreamReader` クラスのオブジェクトで、クライアントが送信したデータの受信処理を司る。また「ライター」は `StreamWriter` クラスのオブジェクトで、クライアントに対する応答の送信処理を司る。

■ StreamReader オブジェクトによるデータの受信

非同期の `read` メソッドによって、相手システムからのデータを受信することができる。

書き方： `await リーダ.read(データ長)`

`StreamReader` オブジェクトの「リーダー」から「データ長」(バイト単位)のサイズのデータを読み取る。この処理は非同期に実行され、受信完了を待機する。`read` メソッドは読み取ったデータをバイト列 (`bytes` 型) として返す。

相手システムが「データ長」よりも小さなデータを送信した場合は、`read` メソッドの結果はそのデータ全体となり、「データ長」よりも大きなデータを送信した場合は、最初の「データ長」の部分だけを返し、残りは受信バッファに残される。受信バッファに残されたデータは次の `read` メソッドの実行時に読み取られる。受信バッファの残部が更に「データ長」より大きい場合は、次の `read` メソッドの実行において、残部のデータの先頭の「データ長」の大きさの部分只得られ、その残部が受信バッファに残される。(以下同様)

相手システムが接続を終了すると、自システム側の `read` メソッドは空のバイト列 `b''` を返す。これを応用すると、相手システム側の終了を検知することができる。

■ StreamWriter オブジェクトによるデータの送信

`write` メソッドによって、相手システムにデータを送信することができる。

書き方： `ライタ.write(データ)`

`StreamWriter` オブジェクトの「ライター」を介して相手システムに「データ」(`bytes` 型)を送信する。`write` メソッドは同期処理であり、この直後に、非同期の `drain` メソッドで送信完了を待機する。

書き方： `await ライタ.drain()`

クライアントとサーバは互いに `read`, `write` メソッドを用いて送受信を繰り返す。一連の通信を終了するにはライターに対して `close` メソッドを実行する。

書き方： `ライタ.close()`

この処理は同期処理であり、終了処理の完了を非同期の `wait_closed` メソッドで待機する。

書き方： `await ライタ.wait_closed()`

■ クライアント側でのサーバへの接続要求

非同期の `open_connection` 関数で、サーバに対して接続要求ができる。

書き方： `await asyncio.open_connection(ホスト名, ポート番号)`

「ホスト名」、「ポート番号」のサーバに接続要求する。接続が確立すると `StreamReader` オブジェクト (リーダー) と `StreamWriter` オブジェクト (ライター) のタプルが返される。

「ホスト名」にはサーバの IP アドレスもしくは FQDN の文字列を与える。「ポート番号」は整数値で与える。

データの送信と受信の処理は、サーバ側の場合と同様に、リーダーに対する `read` メソッド、ライターに対する `write` メ

ソッドで行う。クライアント側での通信の終了の処理もサーバの場合と同様である。

■ 相手システム、自システムに関する情報の取得

ライタ (StreamWriter オブジェクト) に対して `get_extra_info` メソッドを実行することで、相手システムや自システムに関する情報を取得することができる。

書き方: `ライタ.get_extra_info('peername')`

相手システムのホスト名 (あるいは IP アドレス) とポート番号のタプルを返す。 `get_extra_info` の引数に `'sockname'` を与えると、自システムの IP アドレスとポート番号のタプルを返す。

【サンプルプログラム】

asyncio ライブラリのみ使用する形で通信を実現した例を `asyncSocketSv02.py` (サーバ), `asyncSocketCl02.py` (クライアント) に示す。

サーバプログラム: `asyncSocketSv02.py`

```
1 # coding: utf-8
2 import asyncio
3
4 # 個別のクライアントとの通信 (ハンドラ関数)
5 async def handleCl( rd, wt ): # リーダ(rd)とライタ(wt)がシステムから与えられる。
6     addr = wt.get_extra_info('peername') # クライアント情報の取得
7     print(f'Connection from {addr}')
8     while True: # クライアントとの応答サイクル
9         dt = await rd.read(1024) # クライアントからのデータ受信を非同期に待機
10        if not dt: # クライアント側から通信終了を検出すると,
11            break # 応答サイクルを終了する。
12        msg = dt.decode('utf-8')
13        print(f"{addr}: {msg}")
14        res = f"[echo] {msg}".encode('utf-8')
15        wt.write(res) # クライアントへデータ送信 (同期処理)
16        await wt.drain() # 送信処理の完了を非同期に待機する。
17        if msg.strip() == 'end': # クラアントからのデータが'end'なら,
18            break # 応答サイクルを終了する。
19        wt.close() # 接続の終了処理 (同期処理)
20        await wt.wait_closed() # 接続終了を非同期に待機する。
21
22 # サーバの作成と起動
23 async def exe():
24     # サーバインスタンスの生成
25     sv = await asyncio.start_server(handleCl, '127.0.0.1', 8000)
26     try:
27         await sv.serve_forever() # サーバをイベントループ上で起動 (永続稼働)
28     except asyncio.CancelledError: # 強制終了などの検知
29         print('サーバ終了') # 終了メッセージ
30     finally: # サーバ終了時の処理
31         sv.close() # サーバを閉じる処理 (同期処理)
32         await sv.wait_closed() # その処理の完了を非同期に待機する。
33
34 asyncio.run( exe() ) # サーバの起動-終了のタスクをイベントループに投入
```

クライアントプログラム: `asyncSocketCl02.py`

```
1 # coding: utf-8
2 import asyncio
3
4 # クライアント用コルーチン
5 async def client():
6     # サーバに接続要求し, リーダとライタを取得
7     rd, wt = await asyncio.open_connection('127.0.0.1', 8000)
8     while True: # サーバへのデータ送信と応答受信のサイクル
9         msg = input('終了:end> ').rstrip() # コンソール入力の受け付け
10        wt.write( msg.encode('utf-8') ) # サーバにデータ送信 (同期処理)
11        await wt.drain() # データの送信完了を非同期に待機
12        dt = await rd.read(1024) # サーバからの応答受信を非同期に待機
13        print(f"Received: {dt.decode('utf-8')}")
14        if msg == 'end': # コンソール入力がない場合,
15            break # サーバとの受信-送信サイクルを終了
```

```

16     wt.close()                # 接続終了の処理（同期処理）
17     await wt.wait_closed()    # 接続終了処理を非同期に待機
18
19 asyncio.run( client() )      # クライアントのタスクをイベントループに投入

```

これらは先の `asyncSocketSv01.py`, `asyncSocketCl01.py` と同様の通信処理を実現するものであるが、サーバ側の停止処理を例外処理の形で実装している。このため、サーバを `CTRL+C` で終了するとエラーメッセージを出力せずに「サーバ終了」と出力する。

5.1.6.3 通信の処理において注意すべき例外

今回挙げたサンプルプログラムは、基本的な機能の解説を目的として作られているので、例外処理なども十分ではない。特に通信の処理においては様々な例外が発生する可能性があるので、適切に例外処理を実装することが重要である。

通信処理の実装の際に注意すべき例外として基本的なものを表 46 に挙げる。

表 46: 通信の処理で発生する例外（一部）

例 外	解説
<code>ConnectionRefusedError</code>	サーバに接続できなかった場合に発生する。サーバが起動していないことやネットワークの問題が原因である。
<code>TimeoutError</code>	接続やデータの送受信がタイムアウトした場合に発生する。ネットワークの遅延やサーバの応答が遅い場合に発生することがある。
<code>ConnectionResetError</code>	接続がリセットされた場合に発生する。これは、リモートホストが接続を強制的に終了した場合に発生する。
<code>BrokenPipeError</code>	データの送信中に接続が切断された場合に発生する。これは、リモートホストが接続を閉じた後にデータを送信しようとした場合に発生する。
<code>asyncio.CancelledError</code>	非同期タスクがキャンセルされた場合に発生する。タスクのキャンセル後のリソースの解放といった適切な後処理のためにこのハンドリングが必要である。
<code>OSError</code>	一般的な入出力エラー。ネットワークの問題やファイルシステムのエラーなど、様々な原因で発生する。

5.2 WWW コンテンツ解析

WWW は TCP/IP 通信の基礎の上に成り立っているサービスであり、Python と各種のライブラリを使用することで、WWW サーバへのリクエストの送信、コンテンツの取得、コンテンツの解析などが実現できる。ここでは、WWW コンテンツの取得と解析²³⁹ に関する基礎的な事柄について説明する。

5.2.1 requests ライブラリ

requests ライブラリは WWW サーバに対して各種のリクエストを送信したり、WWW サーバからコンテンツを取得するための基本的な機能を提供する。このライブラリはインターネットサイト <http://docs.python-requests.org/> から入手できる。利用に先立って Python システム用にインストールしておく必要があるので、ソフトウェア管理ツール²⁴⁰ を使用してインストールする。requests ライブラリを Python で使用するには次のようにして読み込む。

```
import requests
```

5.2.1.1 リクエストの送信に関するメソッド

■ コンテンツの取得（GET リクエストの送信）

URL を指定してコンテンツを取得するには `get` メソッドを使用する。

書き方： `r = requests.get(コンテンツの URL)`

BASIC 認証を行ってコンテンツを取得するには次のようにする。

書き方： `r = requests.get(コンテンツの URL, auth=(ユーザ名, パスワード))`

■ フォームの送信（POST リクエストの送信）

フォームの内容²⁴¹ を WWW サーバに送信するには `post` メソッドを使用する。このときフォームの項目の「名前」と「値」の対応を Python の辞書型オブジェクト²⁴² にして与える。

書き方： `requests.post(コンテンツの URL, 辞書型オブジェクト)`

これら各メソッドが返すオブジェクトにリクエスト結果の情報が保持されている。

5.2.1.2 取得したコンテンツに関するメソッド

WWW サーバにリクエストを送信すると、それに対するサーバからの応答が返され、それら情報がメソッドの戻り値となる。以後これを**応答オブジェクト**と呼ぶ。

例. ウィキペディアサイトの Python に関する記事の取得

```
import requests

r = requests.get('https://ja.wikipedia.org/wiki/Python')
```

この結果、`r` にサーバからの応答（WWW コンテンツを含む）が応答オブジェクトとして得られる。次に紹介する各種のメソッドを使用することで、取得した応答オブジェクトから様々な情報を取り出すことができる。

■ 取得したコンテンツ全体

WWW サーバから送られてきたコンテンツ全体は応答オブジェクトの `text` プロパティに保持されている。すなわち、

`応答オブジェクト.text`

とすることでコンテンツ全体を取得できる。先の例で得られた応答オブジェクトの `text` プロパティを表示すると次のようになる。

²³⁹ これら一連の処理は**ウェブスクレイピング**と呼ばれている。

²⁴⁰ PSF 版 Python での導入方法に関しては、巻末付録「A.4 PIP によるライブラリ管理」（p.404）を参照のこと。Anaconda の場合は Anaconda Navigator でパッケージ管理を行う。

²⁴¹ `< FORM >... </FORM >` で記述された HTML コンテンツ。

²⁴² INPUT タグの `'NAME='` と `'VALUE='` に指定するものを辞書型オブジェクトにする。

例. `r.text` の表示 (先の例の続き)

```
<!DOCTYPE html>
<html class="client-nojs" lang="ja" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Python - Wikipedia</title>
<script>document.documentElement.className = document.documentElement.className.
replace( /(^|\s)client-nojs(\s|$)/, "$1client-js$2" );</script>
<script>(window.RLQ=window.RLQ||[]).push(function()mw.config.
set("wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,
:
(以下省略)
:
```

■ 応答のステータスの取得

応答オブジェクトの `status_code` プロパティには、リクエスト送信に対する WWW サーバからの応答が保持されている。

例. `r.status_code` の表示 (先の例の続き)

```
200
```

■ エンコーディング情報の取得

応答オブジェクトの `encoding` プロパティには、得られたコンテンツのエンコーディングに関する情報が保持されている。

例. `r.encoding` の表示 (先の例の続き)

```
UTF-8
```

■ ヘッダー情報の取得

応答オブジェクトの `headers` プロパティには、得られたコンテンツのヘッダー情報 (Python の辞書型オブジェクト) が保持されている。

例. `r.headers` の内容 (先の例の続き)

```
{'Date': 'Tue, 09 May 2017 06:22:40 GMT',
 'Content-Type': 'text/html; charset=UTF-8',
 'Content-Length': '46335',
 'Connection': 'keep-alive',
 'Server': 'mw1264.eqiad.wmnet',
:
(途中省略)
:
 'Cache-Control': 'private, s-maxage=0, max-age=0, must-revalidate',
 'Accept-Ranges': 'bytes'}
```

■ Cookie 情報の取得

応答オブジェクトの `cookies` プロパティには、得られたコンテンツの Cookie 情報 (RequestsCookieJar 型オブジェクト²⁴³) が保持されている。

5.2.1.3 Session オブジェクトに基づくアクセス

WWW サーバとの間で情報をやり取りする際のセッション情報は、Session オブジェクトとして扱うことができ、Session オブジェクトに対してコンテンツの取得といったメソッドが使用できる。

例. Session オブジェクトに基づく処理

```
>>> import requests [Enter] ←ライブラリの読み込み
>>> s = requests.Session() [Enter] ← Session オブジェクトの生成
>>> r = s.get('https://ja.wikipedia.org/wiki/Python') [Enter] ←コンテンツの取得
```

この例は Session オブジェクトに基づいてコンテンツの取得を行うものであり、処理の結果として応答オブジェクト `r` が得られている。

²⁴³本書では解説を割愛する。

本書での requests ライブラリに関する解説は以上で終わるが、より多くの情報が配布元サイトや有志の開発者たちによって配信されているのでそれらを参照すること。

requests ライブラリは WWW サーバとの通信における基本的な機能を提供するが、取得した WWW コンテンツの解析といったより高度な処理を行うには、更に別のライブラリを使用した方が良い。次に紹介するライブラリは、取得した WWW コンテンツ（より一般的に XML コンテンツ）の解析や、あるいはコンテンツの構築自体を可能とするものである。

5.2.2 Beautiful Soup ライブラリ

Beautiful Soup は HTML を含む XML 文書の解析や構築を可能とするライブラリであり

<https://www.crummy.com/software/BeautifulSoup/>

から入手することができるので、利用に際して Python システムにインストール²⁴⁴ しておく。また利用するには、次のようにして Python に読み込む。

```
from bs4 import BeautifulSoup
```

以後は Beautiful Soup を BS と略す。

5.2.2.1 BS における HTML コンテンツの扱い

BS では、与えられた HTML (XML) コンテンツを独自のデータ構造である BeautifulSoup オブジェクト（以下 BS オブジェクトと略す）に変換して保持し、それに対して解析や編集の処理を行う。

次のような HTML コンテンツ test15.html がある場合を例に挙げて BS の使用方法を説明する。

HTML コンテンツ：test15.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <title>Pythonに関する情報 </title>
6 </head>
7 <body>
8 <h1>Pythonに関する情報 </h1>
9 <p>下記のリンクをクリックしてPythonに関する紹介を読んでください。 </p>
10 <a href="https://ja.wikipedia.org/wiki/Python">ウィキペディアの記事へ</a>
11 <h2>Beautiful Soupに関する情報 </h2>
12 <p>下記のリンクをクリックしてBeautiful Soupのドキュメントを閲覧してください。 </p>
13 <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/">
14 Beautiful Soupのドキュメント </a>
15 </body>
16 </html>
```

この HTML コンテンツを読み込んで BS オブジェクトに変換する例を次に示す。

例. HTML コンテンツの読み込み

```
>>> from bs4 import BeautifulSoup  [Enter]    ←ライブラリの読み込み
>>> txt = open('test15.html','r',encoding='utf-8').read()  [Enter]    ← HTML ファイルの読み込み
>>> sp = BeautifulSoup(txt,'html5lib')  [Enter]    ← BS オブジェクトの生成
```

この例では、ファイル test15.html の内容を txt に読み込みし、それを BeautifulSoup のコンストラクタの引数に与えて BS オブジェクト sp を生成している。コンストラクタの 2 番目の引数にはコンテンツを解析するためのパーサ（'html5lib' の部分）を指定する。パーサとしてはこの例の html5lib 以外にも lxml などもあるが、それらも使用に際しては別途インストールしておく。

■ BS オブジェクトから文字列への変換（整形あり）

BS オブジェクトに対して prettify メソッドを実行すると、BS オブジェクトの内容を整形して文字列オブジェクトに変換する。

²⁴⁴PSF 版 Python での導入方法に関しては、巻末付録「A.4 PIP によるライブラリ管理」(p.404) を参照のこと。Anaconda の場合は Anaconda Navigator でパッケージ管理を行う。

例. prettify メソッドによる変換

```
>>> print( sp.prettify() )  ←変換して表示
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>
      Python に関する情報
    </title>
  </head>
  <body>
    <h1>
      Python に関する情報
    </h1>
    .
    (途中省略)
    .
    <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/">
      Beautiful Soup のドキュメント
    </a>
  </body>
</html>
```

整形しないテキストは BS オブジェクトのプロパティ contents が保持している。(次の例参照)

BS オブジェクトの contents プロパティ

```
>>> print( sp.contents )
['html', <html><head>
<meta charset="utf-8"/>
<title>Python に関する情報</title>
</head>
:
(途中省略)
:
</body></html>]
```

このようにリストの形式で保持されている。従って HTML のテキストは

BS オブジェクト.contents[1]

として取り出すことができる。

■ 指定したタグの検索

BS オブジェクトに対して指定したタグの検索を実行するには find_all メソッドを使用する。

例. <a>タグを全て取り出す。

```
>>> sp.find_all('a')  ←検索実行
[<a href="https://ja.wikipedia.org/wiki/Python">ウィキペディアの記事へ</a>,
 <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/"> Beautiful Soup のドキュメント</a>]
```

見つかったタグをリストにして返していることがわかる。

■ コンテンツの階層構造

BS オブジェクトの構造は基本的にリストである。従って先に説明した contents プロパティに要素の添え字を付けて更に配下の contents プロパティを取り出すということを繰り返すことで全ての要素にアクセスできる。

contents に添え字を付けて取り出したものは、1つのタグで記述された HTML の文であり、それが持つ name プロパティにはそのタグの名前が保持されている。

例. タグの取り出し

```
>>> sp.contents[1].contents[2].name  ← body タグの取り出し
'body'                                ←処理結果
```

本書での BeautifulSoup ライブラリに関する解説は以上で終わるが、より多くの情報が配布元サイトや有志の開発者たちによって配信されているのでそれらを参照されたい。

6 外部プログラムとの連携

外部のプログラムを Python プログラムから起動する方法について説明する。C 言語や Java といった処理系でプログラムを翻訳、実行する場合と比べると、インタプリタとしての Python 処理系ではプログラムの実行速度は非常に遅い。従って、大きな実行速度を要求する部分の計算処理は、より高速な外部プログラムに委ねるべきである。実際に、科学技術系の計算処理やニューラルネットワークのシミュレーション機能を Python 用に提供するライブラリの多くは外部プログラムとの連携を応用している。

計算速度の問題に限らず、外部の有用なプログラムを呼び出して Python プログラムと連携させることは、高度な情報処理を実現するための有効な手段となる。

Python から外部プログラムを起動するには `subprocess` モジュールを使用する。このモジュールは次のようにして Python 処理系に読み込む。

```
import subprocess
```

6.1 外部プログラムを起動する方法

《 外部プログラムの起動 》

書き方： `subprocess.run(コマンド文字列, shell=True)`

オペレーティングシステム (OS) のコマンドシェルを起動して「コマンド文字列」で与えられたコマンドを実行する。run メソッドはコマンドの終了を待ち、CompletedProcess オブジェクトを返す。コマンド文字列は、コマンド名と引数 (群) を空白文字で区切って並べたものである。

この方法はコマンドシェルを介するので、外部プログラム起動時にシェル変数の設定などが反映される。ただし、シェルも 1 つのプロセスとして起動するので、シェルを介さずに外部プログラムを起動する場合に比べると、システムに対するプロセス管理の負荷が高くなる。シェルを介さずに外部プログラムを起動するには、run メソッドの引数にキーワード引数 `shell=False` を指定する。

例. Windows のコマンド 'dir' を発行する例。

```
>>> import subprocess  [Enter]      ←モジュールの読み込み
>>> subprocess.run('dir *.eps',shell=True)  [Enter]  ← dir コマンドでファイルの一覧表示
Volume in drive C has no label.
Volume Serial Number is 18EE-2F9E

Directory of C:\Users\katsu\TeX\Python

2016/07/23  18:16          530,687 ClientServer.eps
2016/07/23  19:09          511,071 CSsocket0.eps
2016/07/23  19:22          546,418 CSsocket1.eps
2017/05/21  18:10          851,177 Earth_small.eps
2017/04/29  14:23          528,051 HBoxVBox.eps
          ⋮
          (以下省略)
          ⋮
```

これは、Windows のコマンドシェル (コマンドプロンプト) である `cmd.exe` を起動して、その内部コマンド²⁴⁵ である `dir` を実行している例である。呼び出された外部プログラムはサブプロセスとして実行される。

6.1.1 標準入出力の接続

サブプロセス (外部プログラム) の標準入出力に対してデータを送受信する方法について説明する。独立したプログラム同士が通信するための代表的な方法にソケットとパイプがあり、subprocess モジュールはパイプを介した標準入出力の送受信を実現するための簡便な方法を提供している。

²⁴⁵Windows の `cmd.exe` によって解釈されて実行されるコマンド (`cmd.exe` 自体が持つ機能) であり、`dir` 自体は単独の実行形式 (`*.exe`) プログラムではない。

《サブプロセスとのパイプを介した通信》

書き方： `subprocess.Popen(コマンド文字列, shell=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)`

キーワード引数 `stdin`, `stdout`, `stderr` に `subprocess.PIPE` を与えることで、サブプロセスの標準入力、標準出力、標準エラー出力が Python プログラム側からアクセスできるようになる。

`Popen` メソッドを実行すると、戻り値として `Popen` オブジェクトが返され、これのプロパティ `stdin`, `stdout`, `stderr` に対して各種の入出力用メソッドを使用することで、サブプロセスに対して実際にデータを送受信することができる。

【サンプルプログラム】

ここでは、C 言語で作成したシミュレーションプログラムを Python から起動する例を挙げて説明する。

■ シミュレーションプログラム

円の軌跡を描く次のようなダイナミクスをシミュレートするプログラムを考える。

$$\frac{dx}{dt} = -y, \quad \frac{dy}{dt} = x$$

このダイナミクスを離散化すると次のような式となる。

$$\Delta x = -y \cdot \Delta t, \quad \Delta y = x \cdot \Delta t$$

これを実現するプログラムは、

$$x = x - y \cdot dt, \quad y = y + x \cdot dt$$

となり、 x, y に適当なる初期値を与えてこのプログラムを繰り返すとダイナミクスのシミュレーションが実現できる。これを C 言語で記述したものが `sbpr01.c` である。

このプログラムは $(x, y) = (10.0, 0.0)$ を初期値として、時間の刻み幅 dt を 10^{-8} として円の軌跡をシミュレートする。軌跡が円周を一周するとシミュレーションが終了する。

外部プログラム：sbpr01.c

```
1  #include    <stdio.h>
2  #include    <time.h>
3
4  int main()
5  {
6      double   x, y, dt;    /* 座標と微小時間 */
7      long     c;           /* ループカウンタ */
8      clock_t  t1, t2;      /* 時間計測用変数 */
9
10     /* シミュレーションの初期設定 */
11     x = 10.0;  y = 0.0;
12     dt = 0.00000001;      /* 時間間隔 */
13     t1 = clock();         /* 開始時刻 */
14     for ( c = 0L; c < 628318530L; c += 1L ) {
15         y += x * dt;
16         x -= y * dt;
17         /* 中ヌキ出力 */
18         if ( c % 200000L == 0 ) {
19             printf("%16.12lf,%16.12lf\n",x,y);
20         }
21     }
22     t2 = clock();         /* 終了時刻 */
23
24     /* 計算時間の表示（標準エラー） */
25     fprintf(stderr,"%ld (msec)", t2-t1);
26 }
```

このプログラムを翻訳して実行すると、標準出力に次のような形で x, y の値を出力する。


```

10.000000000000, 0.000000100000
9.999979999707, 0.020000086666
9.999919999507, 0.039999993333
9.999819999640, 0.059999739999
9.999680000507, 0.079999246666
9.999500002667, 0.099998433337
:
(以下省略)
:

```

このプログラムからの出力をテキストデータとして保存して gnuplot ²⁴⁶ でプロットした例を図 40 に示す。

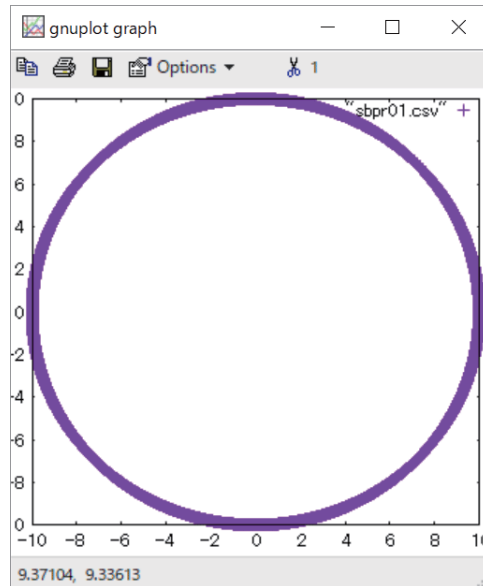


図 40: gnuplot によるプロット例

このシミュレーションは dt を非常に小さく取っており、ダイナミクスが終了するまで x, y の移動計算を 6 億回以上実行する。これは Python のプログラムとして実行するには時間的にも適切ではないため、C 言語で実装した。

次に、このシミュレーションプログラムを Python プログラムから呼び出して、データプロットのためのライブラリ matplotlib ²⁴⁷ を使ってプロットする例を示す。

Python 側のプログラムを sbpr01.py に示す。

プログラム：sbpr01.py

```

1  # coding: utf-8
2  import subprocess
3  import matplotlib.pyplot as plt
4
5  # サブプロセスの生成
6  pr = subprocess.Popen( 'sbpr01.exe', stdout=subprocess.PIPE, shell=True )
7
8  # サブプロセスの標準出力からのデータの受け取り
9  lx = []; ly = []      # これらリストのデータを蓄積
10 buf = pr.stdout.readline().decode('utf-8').rstrip() # 初回読取り (1行)
11 while buf:
12     [bx,by] = buf.split(',')      # CSVの切り離し
13     lx.append( float(bx) ); ly.append( float(by) ) # データの蓄積
14     # 次回読取り (1行)
15     buf = pr.stdout.readline().decode('utf-8').rstrip()
16
17 # matplotlibによるプロット
18 plt.plot(lx, ly, 'o-', label="circle")
19 plt.xlabel("x")
20 plt.ylabel("y")
21 plt.legend(loc='best')
22 plt.show()

```

²⁴⁶オープンソースとして公開されているデータプロットツール。(http://www.gnuplot.info/)

²⁴⁷インターネットサイト <https://matplotlib.org/> でライブラリとドキュメントが公開されている。

このプログラムは Windows 環境で実行するものである。先のシミュレーションプログラムが Windows の実行形式ファイル sbpr01.exe として作成²⁴⁸されており、これと sbpr01.py は同じディレクトリに配置されているものとする。このプログラムを実行すると、シミュレーション結果が図 41 のようなプロットとして表示される。

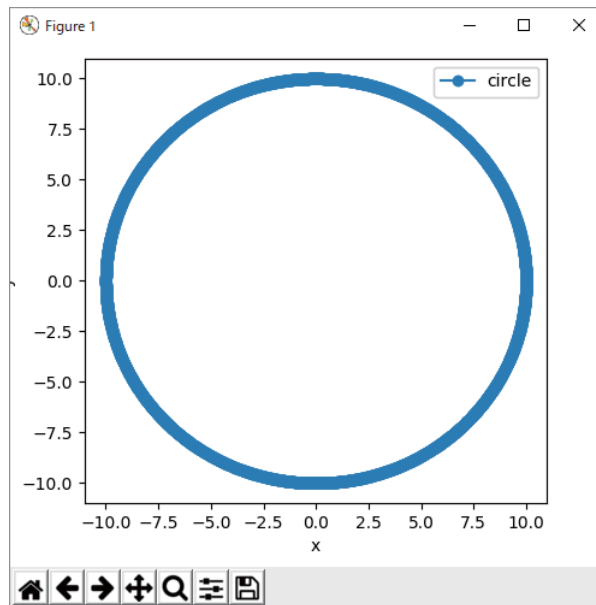


図 41: matplotlib を用いたプロット

6.1.1.1 外部プログラムの標準入力のクローズ

外部プログラムの標準入力への送信を終了してクローズするには close メソッドを用いる。具体的には、サブプロセスオブジェクトの stdin に対して close メソッドを実行する。

例. サブプロセス pr の stdin を閉じる

```
pr.stdin.close()
```

多くのコマンドツールは、標準入力が開じられると終了する。そのようなコマンドツールを Python から起動した場合、この方法で当該サブプロセスを終了することができる。

6.1.2 非同期の入出力

先の例 (sbpr01.py) では、サブプロセスからの出力を受け取る繰り返し処理が終了するまで他の処理はできない。サブプロセスとのパイプを介した入出力を非同期に実行する最も簡単な方法は、入出力の繰り返し処理を独立したスレッドで実行すること²⁴⁹である。ここではサンプルプログラムを示しながらそのための方法を例示する。

【サンプルプログラム】

一定時間が経過する毎に経過時間を表示するプログラム（一種のタイマー）を C 言語で記述したものを sbpr02.c に示す。

外部プログラム：sbpr02.c

```
1  #include    <stdio.h>
2  #include    <time.h>
3
4  int main(ac,av)
5  int ac;
6  char **av;
7  {
8      int      n, c = 1;
9      clock_t d, t1, t2; /* 時間計測用変数 */
10
11     if ( ac < 3 ) {
```

²⁴⁸実行形式ファイルのパス名（ファイル名）などの形式は OS 毎に異なるので注意すること。

²⁴⁹UNIX 系 OS のデーモンプロセスや Windows のサービスのよう、入出力や通信の処理を受け付ける常駐型プログラムを Python で実現するには asyncio ライブラリを使用するのが良い。詳しくは Python の公式サイトを参照のこと。

```

12         fprintf(stderr, "Usage: sbpr02 Duration(ms) Iteration...");
13         return -1;
14     } else {
15         sscanf(av[1], "%ld", &d);
16         sscanf(av[2], "%d", &n);
17     }
18
19     t1 = clock();          /* 開始時刻 */
20     printf("%ld\n", t1);
21     while ( -1 ) {
22         t2 = clock();      /* 現在時刻 */
23         if ( t2 - t1 >= d ) {
24             printf("%ld\n", t2);
25             fflush(stdout); /* 出力バッファをフラッシュ */
26             t1 = t2;
27             c++;
28             if ( c > n ) {
29                 break;
30             }
31         }
32     }
33
34     fprintf(stderr, "%s %s %s: finished.\n", av[0], av[1], av[2]);
35 }

```

このプログラムは、起動時の第1引数に経過時間 (ms) を、第2引数に計測回数を与えるものである。また経過時間を標準出力に、終了のメッセージを標準エラー出力に出力する。このプログラムを翻訳して実行した例を次に示す。

```

C:\¥Users¥katsu¥Python> sbpr02 3000 2          ← 3秒経過を2回通知する指定
0                                                  ←経過時間を標準出力に出力
3001
6001
sbpr02 3000 2: finished.                        ←終了メッセージを標準エラー出力に出力

```

これを外部プログラムとして同時に複数並行して起動するプログラムの例を sbpr02.py に示す。

プログラム：sbpr02.py

```

1  # coding: utf-8
2  import subprocess
3  from threading import Thread
4
5  # サブプロセスの生成
6  pr1 = subprocess.Popen( 'sbpr02.exe 3000 3', stdout=subprocess.PIPE, shell=True )
7  pr2 = subprocess.Popen( 'sbpr02.exe 2250 4', stdout=subprocess.PIPE, shell=True )
8
9  # pr1からの入力を受け取る関数
10 def Exe1():
11     while True:
12         buf = pr1.stdout.readline().decode('utf-8').rstrip()
13         if buf:
14             print('プロセス1:', buf)
15         else:
16             break
17
18 # pr2からの入力を受け取る関数
19 def Exe2():
20     while True:
21         buf = pr2.stdout.readline().decode('utf-8').rstrip()
22         if buf:
23             print('プロセス2:', buf)
24         else:
25             break
26
27 # サブプロセスの入力を受け付けるスレッド
28 th1 = Thread( target=Exe1, args=() )
29 th2 = Thread( target=Exe2, args=() )
30
31 # スレッドの起動
32 print('*** 実行開始 ***')

```

```

33 | th1.start()
34 | th2.start()
35 |
36 | # スレッド終了の待ち受け
37 | th1.join()
38 | th2.join()
39 | print('*** 実行終了 ***')
```

解説：

6,7 行目で外部プログラムを 2 つサブプロセスとして生成して、それぞれ pr1, pr2 としている。それらの標準出力をパイプ経由で取得する処理を、関数 Exe1, Exe2 として定義（10～16 行目）している。それら関数を独立したスレッドとして生成（28,29 行目）して起動（33,34 行目）している。それらスレッドは同時に並行して別々に動作する。37,38 行目では、2 つのスレッドが終了するのを待機している。スレッドの扱いに関しては「4.4 マルチスレッドとマルチプロセス」（p.255）を参照のこと。

このプログラムを実行した例を示す。

```

*** 実行開始 ***
プロセス 2:  0
プロセス 2: 2251
プロセス 1:  0
プロセス 1: 3000
プロセス 2: 4504
プロセス 1: 6000
プロセス 2: 6768
sbpr02.exe 3000 3: finished.
プロセス 1: 9000
プロセス 2: 9031
sbpr02.exe 2250 4: finished.
*** 実行終了 ***
```

各プロセスからの出力（経過時間）が標準出力に、各プロセスからの終了メッセージが標準エラー出力に出力されている²⁵⁰。

6.1.3 外部プロセスとの同期（終了の待機）

Popen メソッドで生成されたプロセスが終了するまで待機するには wait メソッドを使用する。すなわち、Popen で生成されたプロセス pr の終了に同期して、終了するまで処理をブロックする（終了まで待機する）には、

```
pr.wait()
```

とする。

6.1.4 外部プログラムを起動する更に簡単な方法

外部プログラムと呼び出し側の Python のプログラムの間で標準入出力の接続をしない場合には、os モジュールの system 関数を使って更に簡単に外部プログラムを起動することができる。

書き方： os.system(' 外部プログラムを起動するためのコマンドライン')

これを Windows の環境で実行した例を示す。

²⁵⁰ 出力のリダイレクトをしない場合、標準出力、標準エラー出力とも同じターミナルウィンドウになり、両方が混在した形で表示される..

例. dir コマンドの実行 (Windows 環境)

```
>>> import os  ← os モジュールの読み込み
>>> os.system( 'dir *.pdf' )  ← dir コマンドの発行
Volume in drive C has no label. ←実行結果の表示
Volume Serial Number is 18EE-2F9E
Directory of C:\Users\katsu
2017/09/23 13:37 38,937 lab_rep_speed1.pdf
2018/09/22 12:07 354,666 python.ai_note.pdf
2018/09/16 14:30 4,206,175 python_main.pdf
2018/09/14 20:48 5,420,705 python_modules.pdf
2018/05/12 14:47 3,640,064 python_stat.pdf
5 File(s) 20,829,517 bytes
0 Dir(s) 54,815,989,760 bytes free
0 ← os.system 関数の戻り値
```

参考. この方法で外部プログラムを起動する場合、データの受け渡しはファイルを介すると良い。例えば、先の例において、dir コマンドの出力結果を Python プログラム側で受け取るには次のような方法がある。

例. dir コマンドの実行結果をファイルを通じて受け取る (Windows 環境)

```
>>> import os  ← os モジュールの読み込み
>>> os.system('dir *.pdf > temp.txt')  ←標準出力をファイル 'temp.txt' に保存
0 ← os.system 関数の戻り値
>>> f = open('temp.txt','r')  ←ファイル 'temp.txt' を開く251
>>> t = f.read()  ←ファイルの内容を読み取って t に受け取る
>>> print( t )  ← t の内容を表示
Volume in drive C has no label. ←ファイルの内容の表示
Volume Serial Number is 18EE-2F9E
Directory of C:\Users\katsu
2017/09/23 13:37 38,937 lab_rep_speed1.pdf
2018/09/22 12:07 354,666 python.ai_note.pdf
2018/09/16 14:30 4,206,175 python_main.pdf
2018/09/14 20:48 5,420,705 python_modules.pdf
2018/05/12 14:47 3,640,064 python_stat.pdf
5 File(s) 20,829,517 bytes
0 Dir(s) 54,815,989,760 bytes free
```

²⁵¹open 関数には引数 'encoding=' を与えてファイルのエンコーディングを指定しなければならない場合がある。

7 サウンドの入出力

ここでは基本的なサウンドデータの取り扱い方法について説明する。主な内容は

1. データ（ファイル）としてのサウンドの入出力
2. リアルタイムのサウンド入力と再生

の2つである。

7.1 基礎知識

この章の内容を理解するに当たり必須となる知識を次に挙げる。

・量子化ビット数

音の最小単位を表現するビット数であり、ある瞬間の音の大きさを何ビットで表現するかを意味する。量子化ビット数が大きいほど取り扱う音の質（音質）が良い。一般的な音楽 CD などでは量子化ビット数は 16（2 バイト）である。

・サンプリング周波数（サンプリングレート）

連続的に変化するアナログの音声をサンプリングによって離散化してデジタル情報として扱うが、1 秒間に音声を何回サンプリングするか（1 秒の音声を何個に分割するか）がサンプリング周波数である。この数値が大きいほど扱う音声の音質が良くなる。一般的な音楽 CD などではサンプリング周波数は 44.1KHz であり、1 秒の音声を 44,100 個に分割している。

・チャンネル数

同時に取り扱う音声の本数がチャンネル数である。日常的に鑑賞するオーディオはアナログ／デジタルの違いに関わらず、左右の音声を別々に扱っており、左右それぞれのスピーカーから再生されることが一般的である。この場合は「左右合計で 2 チャンネルの音声」を扱っていることになる。高級なオーディオセットでは 2 チャンネル以上の音声の取り扱いが可能なものもある。通常の場合ステレオ音声は 2 チャンネル、モノラル音声は 1 チャンネルの扱いを意味する。

7.2 WAV 形式ファイルの入出力：wave モジュール

wave モジュールを使用することで、WAV 形式²⁵²のサウンドデータをファイルから読み込んだり、ファイルに書き出すこと²⁵³ができる。（本書では整数で波形を表現する形の WAV 形式ファイルについて説明する²⁵⁴）

WAV 形式ファイルの取り扱いも通常のファイルの場合と基本的には同じであり、

1. WAV 形式ファイルをオープンする
2. WAV 形式ファイルから内容を読み込む（あるいは書き込む）
3. WAV 形式ファイルを閉じる

という流れとなる。wave モジュールを使用するには次のようにしてモジュールを読み込む。

```
import wave
```

7.2.1 WAV 形式ファイルのオープンとクローズ

通常のファイルの取り扱いと同様に、WAV 形式ファイルを開く場合も「読み込み」もしくは「書き込み」のモードを指定する。開く場合には wave モジュールの open を、閉じる場合は close を使用する。

《 WAV 形式ファイルのオープン 》

書き方： `wave.open(ファイル名, モード)`

WAV 形式ファイルのファイル名（パス名）を文字列で与え、モードには 'rb'（読み込み）か 'wb'（書き込み）を指定する。open 関数が正常に終了すると「読み込み」の場合は Wave_read オブジェクトが、「書き込み」の場合は Wave_write オブジェクトが返され、それらに対して音声データの読み書きを行う。

²⁵²Microsoft 社と IBM 社が開発した音声フォーマットである。音声データの圧縮保存にも対応しているが、非圧縮の PCM データを扱うことが多い。本書では非圧縮のものを前提とする。

²⁵³wave モジュールはサウンドデータの解析などに使用するものであり、音声を録音・再生する機能はない。録音や再生には別のライブラリ（後述の PyAudio など）を使用する。

²⁵⁴32-bit floating-point の WAV 形式ファイルの扱いについては拙書「Python3 ライブラリブック」で解説しています。

読み書きの処理を終える際は close メソッドを Wave_read オブジェクトや Wave_write オブジェクトに対して実行する。

書き方： Wave_read/Wave_write オブジェクト.close()

7.2.1.1 WAV 形式データの各種属性について

WAV 形式ファイルを開いて生成した Wave_read オブジェクトから基本的な属性情報を取り出すには表 47 のようなメソッドを使用する。

表 47: Wave_read オブジェクト wf から情報の取得

メソッド	得られる情報
wf.getnchannels()	チャンネル数
wf.getsampwidth()	量子化ビット数をバイト表現にした値
wf.getframerate()	サンプリング周波数 (Hz)
wf.getnframes()	ファイルの総フレーム数

フレームについて

wave モジュールで WAV 形式データを読み込む場合、音声再生の単位となる**フレーム**（サンプリングにおける 1 時点の値の全チャンネルのセット）の数を指定する。すなわち「n フレームを WAV 形式ファイルから読み込む」という形の扱いとなる。表 47 の中にある**総フレーム数**は当該ファイルを構成する全てのフレームの数である。

7.2.2 WAV 形式ファイルからの読み込み

Wave_read オブジェクトからフレームデータを読み込むには readframes メソッドを使用する。

《 フレームデータの読み込み 》

書き方： Wave_read オブジェクト.readframes(フレーム数)

引数には読み込むフレーム数を指定する。実際のファイルに指定したフレーム数がなければ存在するフレームのみの読み込みとなる。このメソッドの実行により読み込まれた一連のフレームがバイト列として返される。

7.2.3 サンプルプログラム

Wave_read オブジェクトから各種の属性情報を取得するプログラム wave00.py を次に示す。

プログラム：wave00.py

```
1 # coding: utf-8
2 import wave
3
4 # WAV形式ファイルのオープン
5 wf = wave.open('sound01.wav', 'rb')
6 print( 'チャンネル数:\t\t', wf.getnchannels() )
7 print( '量子化ビット数:\t\t', 8*wf.getsampwidth(), '\t(bit)' )
8 print( 'サンプリング周波数:\t', wf.getframerate(), '\t(Hz)' )
9 print( 'ファイルのフレーム数:\t', wf.getnframes() )
10
11 b = wf.readframes(1)
12 print( '1フレームのサイズ:\t', len(b), '\t(bytes)' )
13
14 wf.close()
```

このプログラムを実行した例を次に示す。

チャンネル数:	1	
量子化ビット数:	16	(bit)
サンプリング周波数:	22050	(Hz)
ファイルのフレーム数:	1817016	
1 フレームのサイズ:	2	(bytes)

7.2.4 量子化ビット数とサンプリング値の関係

WAV 形式では、量子化ビット数として 8 ビット（1 バイト）か 16 ビット（2 バイト）のサンプリングデータを扱うことが多い。またそのようなサンプリングデータを符号付きの整数値として扱う場合、表 48 のような範囲の値となる。

表 48: サンプリング値を符号付きの整数で表現する場合の値の範囲

量子化	データ型	解説
8 ビット（1 バイト）	符号付き 1 バイト整数	-128～127
16 ビット（2 バイト）	符号付き 2 バイト整数	-32768～32767

WAV 形式の音声データを表 48 のような形式で出力する際は、この範囲の整数値として調整する必要がある。また入力の際は、この範囲の整数値として波形データを取得する。

7.2.5 読み込んだフレームデータの扱い

WAV 形式ファイルから取得したフレームデータは適切な型のデータに変換することで音声データ解析などに使用することができる。数値データ解析のためのライブラリである `numpy` を用いて音声データを解析用の配列データに変換する例をサンプルプログラム `wave01.py` に示す。

プログラム：wave01.py

```
1  # coding: utf-8
2  import wave
3  import numpy
4
5  # WAV形式ファイルのオープン
6  wf = wave.open('sound01.wav', 'rb')
7
8  # 各種属性の取得
9  chanel = wf.getnchannels()
10 qbit = 8*wf.getsampwidth()
11 freq = wf.getframerate()
12 frames = wf.getnframes()
13
14 b = wf.readframes(frames)
15
16 data = numpy.frombuffer(b, dtype='int16')
17 print( 'データタイプ:', type(data))
18 print( 'データ数:', len(data) )
19
20 #####
21 #   音声の波形データが data に格納されています。           #
22 #   波形の振幅は  -32768   ～   32767   です。             #
23 #   このデータを numpy をはじめとするパッケージで         #
24 #   解析処理することができます。                         #
25 #####
26
27 wf.close()
```

このプログラムの 16 行目でフレームデータを `numpy` の `ndarray` オブジェクトに変換しており、解析が可能となる。ステレオ（2 チャンネル）音声の場合は

[左, 右, 左, 右, …]

の順序で数値が格納されている。フレームデータを展開するには「4.11 バイナリデータの作成と展開：struct モジュール」（p.313）で説明した方法もある。

※ `numpy` ライブラリについての解説は他の情報源に譲る²⁵⁵。

先のプログラム `wave01.py` を少し拡張した、WAV 形式データの波形をプロットするプログラム `wave01-2.py` を次に示す。

²⁵⁵拙書「Python3 ライブラリブック」で解説しています。

プログラム：wave01-2.py

```
1 # coding: utf-8
2 import wave
3 import numpy
4 import matplotlib.pyplot as plt
5
6 # WAV形式ファイルのオープン
7 wf = wave.open('aaa.wav', 'rb')
8
9 # 各種属性の取得
10 chanel = wf.getnchannels()
11 qbit = 8*wf.getsampwidth()
12 freq = wf.getframerate()
13 frames = wf.getnframes()
14
15 # データをnumpyの配列に変換
16 b = wf.readframes(frames)
17 d = numpy.frombuffer(b, dtype='int16')
18 d_left = d[0::2] # 左音声
19 d_right = d[1::2] # 右音声
20 t = [i/freq for i in range(len(d_left))] # 時間軸の生成
21
22 # WAV形式ファイルのクローズ
23 wf.close()
24
25 # matplotlibによるプロット
26 (fig, ax) = plt.subplots(2, 1, figsize=(10,5))
27 plt.subplots_adjust(hspace=0.6)
28 ax[0].plot(t, d_left, linewidth=1)
29 ax[0].set_title('left')
30 ax[0].set_ylabel('level')
31 ax[0].set_xlabel('t (sec)')
32 ax[0].grid(True)
33 ax[1].plot(t, d_right, linewidth=1)
34 ax[1].set_title('right')
35 ax[1].set_ylabel('level')
36 ax[1].set_xlabel('t (sec)')
37 ax[1].grid(True)
38 plt.show()
```

解説

17行目でWAV形式のデータをnumpyの配列データに変換し、更に18-19行名で左右のデータとして分離し、最終的にmatplotlibライブラリによって可視化している。このプログラムを実行して表示されたグラフの例を図42に示す。

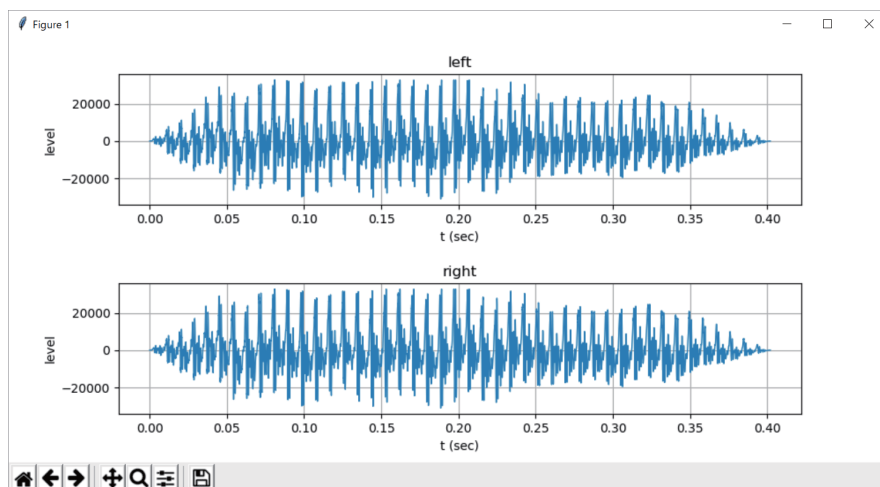


図 42: WAV 形式データのプロット

※ matplotlib ライブラリについての解説は他の情報源²⁵⁶に譲る。

²⁵⁶拙書「Python3 ライブラリブック」で解説しています。

7.2.6 WAV 形式データを出力する例 (1)：リストから WAV ファイルへ

WAV 形式データを出力するには Wave_write オブジェクトに対して各種の属性情報を設定（表 49 参照）し、バイナリデータとして作成した波形データを出力する。

表 49: Wave_write オブジェクトへの属性情報の設定

Wave_write wf に対するメソッド	設定する属性情報
wf.setnchannels(n)	n=チャンネル数
wf.setsampwidth(n)	n=量子化ビット数をバイト表現にした値
wf.setframerate(f)	f=サンプリング周波数 (Hz)

1KHz の正弦波形のデータをリストの形式で生成して、それを WAV 形式データとしてファイルに保存するプログラム wave02.py を次に示す。

プログラム：wave02.py

```
1  # coding: utf-8
2  import wave
3  import struct
4  from math import sin
5
6  #####
7  # 1KHzの正弦波サウンドを44.1KHzでサンプリング      #
8  # する際の sin関数の定義域の刻み幅                  #
9  #####
10 dt = (3.14159265359*2)*1000 / 44100
11
12 #####
13 # 10秒間の正弦波データを生成                        #
14 #####
15 t = 0.0
16 dlist = []
17 for x in range(441000): # 10秒間のデータ
18     y = int(32767.0 * sin(t))
19     dlist.append(y)
20     t += dt
21
22 # 数値のリストをバイナリデータに変換
23 data = struct.pack('h'*len(dlist), *dlist)
24 print('データサイズ:', len(data), 'バイト')
25
26 #####
27 # WAV形式で保存                                      #
28 #####
29 # WAV形式ファイルの作成（オープン）
30 wf = wave.open('out01.wav', 'wb')
31
32 # 各種属性の設定
33 wf.setnchannels(1)      # チャンネル数=1
34 wf.setsampwidth(2)      # 量子化ビット数=2バイト（16ビット）
35 wf.setframerate(44100)  # サンプリング周波数=44.1KHz
36
37 # ファイルへ出力
38 wf.writeframesraw(data)
39
40 wf.close()
```

解説

10～20 行目の部分で正弦波形のデータをリスト dlist として作成している。それを 23 行目でバイナリデータ data に変換している。変換には struct モジュールを使用している。この部分に関しては、「4.11 バイナリデータの作成と展開：struct モジュール」（p.313）、「2.8.1.4 関数呼び出し時の引数に「*」を記述する方法」（p.139）を参照のこと。

30 行目で WAV 形式ファイル 'out01.wav' を作成して、33～35 行目の部分で WAV 形式として必要な各種の属性情報を設定している。実際の出力は 38 行目で行っており、バイト列をそのまま WAV 形式データとして出力するためのメソッド writeframesraw を使用している。WAV 形式データの出力にはこの他にも writeframes メソッドもある。

writeframesraw と writeframes の違いは、出力するフレーム数の設定に関することであるが、詳しくは公式サイト
のドキュメントを参照のこと。

WAV 形式の属性情報の設定には setparams メソッドを使用することもでき、33～35 行目の部分を次のように短く
記述することもできる。

```
wf.setparams( (1,2,44100,len(dlist),'NONE','not compressed') )
```

引数に与えるタプルは次のような 6 つの要素を持つ。

- | | | | |
|---------|--------------|---------|--------------------|
| 1 番目の要素 | - チャンネル数, | 2 番目の要素 | - 量子化ビット数のバイト値, |
| 3 番目の要素 | - サンプリング周波数, | 4 番目の要素 | - 出力フレームの個数, |
| 5 番目の要素 | - 'NONE', | 6 番目の要素 | - 'not compressed' |

7.2.7 WAV 形式データを出力する例 (2) : NumPy の配列から WAV ファイルへ

音声データの解析や合成には、数値演算のためのライブラリである NumPy を使用することが多い²⁵⁷。ここでは、
NumPy の配列として作成した波形データを WAV ファイルに出力する方法について例を挙げて説明する。サンプル
のプログラム wave02-2.py を示す。

プログラム：wave02-2.py

```
1  # coding:utf-8
2  import wave
3  import numpy      # NumPyライブラリ
4  #####
5  # 周波数1KHzの正弦波をサンプリング周波数44.1KHzで10秒の長さで作成する  #
6  #####
7  r = 44100          # サンプリング周波数 (Hz)
8  f = 1000          # サウンドの周波数 (Hz)
9  T = 10            # サウンドの長さ (秒)
10 # 刻み幅 (秒)
11 dt = 1 / r
12 # 波形の作成 (NumPyの配列)
13 x = numpy.arange(0,T,dt)      # 時間軸
14 w = 2.0 * numpy.pi * f      # ω
15 y = 32767.0 * numpy.sin(w*x)  # 波形データ (振幅を-32767～32767に調整)
16 # 要素を2バイト整数に変換して波形データを保存用のバッファにする
17 b = y.astype(numpy.int16).tobytes()
18 #####
19 # WAVファイルとして保存する  #
20 #####
21 wf = wave.open('out01-2.wav','wb') # ファイルを出力用に開く
22 wf.setnchannels(1)                # チャンネル数=1 (モノラル)
23 wf.setsampwidth(2)                # 量子化ビット数=2バイト=16ビット
24 wf.setframerate(r)                 # サンプリング周波数=44.1KHz
25 wf.writeframesraw(b)               # ファイルに書き出し
26 wf.close()                         # ファイルを閉じる
```

解説

このプログラムは、正弦波の波形 (1KHz, 10 秒間) を WAV ファイル 'out01-2.wav' に出力するものである。7～
15 行目で波形データが NumPy の配列 (ndarray) データとして y に得られている。更に 17 行目で配列 y をバイト
列 (bytes 型) のデータに変換して変数 b に与えている。(NumPy の astype, tobytes メソッドを使用)

WAV ファイルへの出力 (21～26 行目) は、先に解説した 'wave02.py' と同様の方法を用いる。また、WAV 形式
の属性情報の設定には setparams メソッドを使用することもでき、22～24 行目の部分を次のように短く記述するこ
とができる。

```
wf.setparams( (1,2,r,len(y),'NONE','not compressed') )
```

7.2.8 サウンドのデータサイズに関する注意点

本書では解説を簡単にするために、WAV 形式ファイルの入出力を 1 度で行っている。すなわち、音声ファイルの
全フレームを一度に読み込んだり、全フレームを一度でファイルに書き込むという形を取った。実用的なオーディオ

²⁵⁷NumPy については拙書「Python3 ライブラリブック」で解説しています。

アプリケーションでは扱う音声データが大きい（数十 MB～数百 MB）場合が多く、全フレームを一度に読み書きするのはシステムの記憶資源の関係上、現実的でないことが多い。実際には使用するシステム資源から判断して「現実的な入出力の単位」のサイズを定めて、そのサイズの入出力を繰り返すという形でサウンドデータを取り扱うことが望ましい。これに関する具体的な方法については、「7.3.2 WAV 形式サウンドファイルの再生」, 「7.3.3 音声入力デバイスからの入力」の所で実装例を挙げて示す。

7.3 サウンドの入力と再生：PyAudio ライブラリ

PyAudio ライブラリを利用することで、リアルタイムの音声入出力ができる。先に説明した wave モジュールは WAV 形式の音声ファイルを取り扱うための基本的な機能を提供するものであり、これと PyAudio を併せて利用することでサウンドの入出力とファイル I/O が実現できるので、実用的なサウンドの処理が可能となる。

PyAudio は PortAudio ライブラリの Python バインディングである。PortAudio はクロスプラットフォーム用のオープンソースのサウンドライブラリであり、開発と保守は PortAudio community によって支えられている。PyAudio の利用に際しては、予め PortAudio をインストールしておく必要²⁵⁸ がある。PortAudio に関する情報はインターネットサイト <http://www.portaudio.com/> を参照のこと。

PSF 版 Python では PyAudio ライブラリは PIP で導入 (p.404 「A.4 PIP によるライブラリ管理」を参照) することができる。Anaconda の場合は Anaconda Navigator でパッケージ管理を行う。また、PyAudio に関して本書で解説していない情報については PyAudio のドキュメントサイト <https://people.csail.mit.edu/hubert/pyaudio/docs/> を参照のこと。

PyAudio を使用するには、次のようにしてライブラリを読み込んでおく。

```
import pyaudio
```

7.3.1 ストリームを介したサウンド入出力

PyAudio では、サウンドの入力源や再生のための出力先をストリームとして扱う。すなわち、サウンドの入力はストリームからデータを取得する形で行い、サウンドの再生はストリームにデータを書き込む形で行う。

【PyAudio オブジェクト】

PyAudio を用いたサウンド入出力は **PyAudio オブジェクト** を基本とする。PyAudio の使用に際しては予め PyAudio オブジェクトを生成しておく。

《 PyAudio オブジェクトの生成 》

書き方： `pyaudio.PyAudio()`

この結果 PyAudio オブジェクトが生成される。

実行例： `p = pyaudio.PyAudio()`

実行結果として PyAudio オブジェクト `p` が得られる。

PyAudio オブジェクトに対して `open` メソッドを使用することでストリームが得られる。`open` メソッドの引数にサウンドの取り扱いに関する属性情報などを与える。

《 ストリームの生成 》

書き方： `PyAudio オブジェクト.open(channels=チャンネル数, rate=サンプリング周波数, format=量子化に関する情報, input=入力機能を使用するか, output=出力機能を使用するか)`

これを実行した結果ストリームが得られる。キーワード引数 `input`, `output` には真理値を指定する。`input`, `output` に同時に `True` を設定することができ、その場合は「入出力両用」のストリームが得られる。

キーワード引数 `format` には次のようにして生成した値を指定する。

PyAudio オブジェクト.get_format_from_width(バイト数)

バイト数には量子化ビット数をバイト単位で表現した整数値を指定する。

ストリームの使用を終える場合は、ストリームのオブジェクトに対して `stop_stream` メソッドを実行し、更に `close` メソッドを実行する。また PyAudio の使用を終える場合は、PyAudio オブジェクトに対して `terminate` メソッドを実行する。

²⁵⁸Windows の版によってはこの処理が必要ない場合がある。macOS の場合は `brew` などの管理ツールで PortAudio をインストールしておく。

7.3.2 WAV 形式サウンドファイルの再生

WAV 形式ファイルから読み込んだフレームデータをシステムのサウンド出力で再生する方法についてプログラム例を挙げて説明する。まずは**ブロッキングモード**と呼ばれる素朴な形での再生（プログラム pyaudio01.py）について説明する。

プログラム：pyaudio01.py

```
1  # coding: utf-8
2  import wave
3  import pyaudio
4  #####
5  # WAV形式サウンドファイルの読み込み #
6  #####
7  # ファイルのオープン
8  wf = wave.open('sound01.wav', 'rb')
9  # 属性情報の取得
10 ch = wf.getnchannels()      # チャネル数
11 qb = wf.getsampwidth()     # 量子化ビット数 (バイト数)
12 fq = wf.getframerate()     # サンプリング周波数
13 frames = wf.getnframes()   # ファイルの総フレーム数
14 # 内容の読み込み
15 buf = wf.readframes(frames)
16 # ファイルのクローズ
17 wf.close()
18 #####
19 # PyAudioによる再生 #
20 #####
21 # PyAudioオブジェクトの生成
22 p = pyaudio.PyAudio()
23 # ストリームの生成
24 stm = p.open( format=p.get_format_from_width(qb),
25               channels=ch, rate=fq, output=True )
26 # ストリームへのサウンドデータの出力
27 stm.write(buf)
28 # ストリームの終了処理
29 stm.stop_stream()
30 stm.close()
31 # PyAudioオブジェクトの廃棄
32 p.terminate()
```

解説

8～17行目でWAV形式サウンドデータを取得して、バイトデータ buf として保持している。生成したストリーム stm に対して write メソッドを使用（27行目）して buf の内容を書き込んでいる。

pyaudio01.py（ブロッキングモードの再生）では、サウンドの再生がメインプログラムのスレッドで実行されるので、サウンドの再生が終了するまで write メソッド部分でプログラムの流れがブロックされる。（待たされる）
実用的なアプリケーションでは、サウンド再生はメインプログラムとは別のスレッドで実行されるべきであり、サウンド再生中もメインプログラムの流れはブロックされるべきではない。次にサウンド再生を別のスレッドで実行する方法について説明する。

【コールバックモードによる再生の考え方】

ストリームへのサウンド出力処理には**コールバックモード**と呼ばれる形態がある。これは、WAV形式ファイルからの音声フレームの入力と、それをストリームに出力するサイクルを**コールバック関数**という形で定義しておき、PyAudioがこの関数を別スレッドで実行するものである。コールバック関数の呼び出しは、WAV形式ファイルから読み込むべきフレームが無くなるまで自動的に繰り返される。コールバック関数は次のような形で定義する。

《 コールバック関数の定義 》

書き方： def 関数名 (入力データ, フレーム数, 時間に関する情報, フラグ) :
 (WAV形式ファイルから「フレーム数」だけ読み込む処理)
 return (読み込んだバイトデータ, pyaudio.paContinue)

このように定義した関数を PyAudio が自動的に繰り返し呼び出す。「入力データ」にはストリームからサウンドを入力する際に取得したバイトデータが与えられる。(再生処理の場合はこの仮引数は無視する)

「フレーム数」は一度に読み取るべきフレームの数であり、コールバック関数を呼び出す際に PyAudio がこの引数に適切な値を与える。「時間に関する情報」「フラグ」に関しては説明を省略する。(詳しくは PyAudio のドキュメントサイトを参照のこと)

この関数は、WAV 形式ファイルから readframes メソッドを使用して読み込んだデータと、pyaudio.paContinue という値をタプルにして返す (return する) ものとして定義する。pyaudio.paContinue に関する説明は省略する。(詳しくは PyAudio のドキュメントサイトを参照のこと)

定義したコールバック関数は、ストリームを生成する段階で、open メソッドにキーワード引数

stream_callback=関数名

として与えることでストリームに登録する。ストリームに対して start_stream メソッドを使用することでサウンドの再生が開始する。

コールバック関数の呼び出しによる方法でサウンドを再生するプログラム pyaudio02.py を次に示す。

プログラム：pyaudio02.py

```
1  # coding: utf-8
2  import wave
3  import pyaudio
4
5  #####
6  # WAV形式サウンドファイルの読み込み #
7  #####
8  # ファイルのオープン
9  wf = wave.open('sound01.wav', 'rb')
10 # 属性情報の取得
11 ch = wf.getnchannels()      # チャンネル数
12 qb = wf.getsampwidth()     # 量子化ビット数 (バイト数)
13 fq = wf.getframerate()     # サンプリング周波数
14
15 #####
16 # PyAudioによる再生 #
17 #####
18 #---- 再生用コールバック関数 (ここから) -----
19 def sndplay(in_data, frame_count, time_info, status):
20     buf = wf.readframes(frame_count)
21     return ( buf, pyaudio.paContinue )
22 #---- 再生用コールバック関数 (ここまで) -----
23
24 # PyAudioオブジェクトの生成
25 p = pyaudio.PyAudio()
26 # ストリームの生成
27 stm = p.open( format=p.get_format_from_width(qb),
28               channels=ch, rate=fq, output=True,
29               stream_callback=sndplay )
30
31 # コールバック関数による再生を開始
32 stm.start_stream()
33
34 # 待ちループ: 「exit」と入力すれば終了処理に進む
35 while stm.is_active():
36     cmd = input('終了->exit: ')
37     if cmd == 'exit':
38         print('終了します...')
39         break
40
41 # ストリームの終了処理
42 stm.stop_stream()
43 stm.close()
44 # PyAudioオブジェクトの廃棄
45 p.terminate()
46
47 # WAVファイルのクローズ
48 wf.close()
```

解説

19～21 行目がコールバック関数 `sndplay` の定義である。この関数は、入力元の WAV 形式データファイルに対応する `Wave.read` オブジェクト（プログラム中の `wf`）に `readframes` メソッドを実行して、音声データを取り出すものである。29 行目ではこの関数をストリームに登録している。32 行目でサウンド再生のコールバック処理が開始し、メインプログラムは更に下の行の実行に移る。

35～39 行目はコマンド待ちのループであり、標準入力に対して `exit` と入力することで、42 行目以降の終了処理に進む。

7.3.2.1 サウンド再生の終了の検出

コールバックモードによるサウンドの再生において、`Wave.read` オブジェクトからの読み込みが終了（ファイル末尾に到達）したことを検出するには、ストリームに対して `is_active` を実行する。この結果、読み込みが終了していれば `False` を、終了していなければ `True` を返す。これは、サウンド再生の終了の検出のための基本的な方法となる。プログラム `pyaudio02.py` の 35 行目でもこれを応用してサウンド再生の終了を検知している。

読み込みが一度終了した `Wave.read` オブジェクトに対して `rewind` メソッドを実行すると、サウンドの再生位置をファイルの先頭に戻すことができる。これを応用すると、サウンドの繰り返し再生が可能となる。具体的には再生が終了したストリームに対して `stop_stream` メソッドを実行し、再生対象の `Wave.read` オブジェクトに対して `rewind` を実行した後に再度ストリームに対して `start_stream` メソッドを実行する流れとなる。

7.3.3 音声入力デバイスからの入力

ここでは、システムに接続されたマイクやライン入力などの音声入力デバイスからリアルタイムにサウンドを入力する方法²⁵⁹ について説明する。

サウンドの入力に関しても、`PyAudio` オブジェクトから生成したストリームオブジェクトを介して処理をする。ストリームの生成に関してもサウンド再生の場合とよく似ており、異なるのは `open` メソッドの引数に

```
input=True
```

というキーワード引数を与えることと、1 度の入力で受け取るフレーム数を指定する

```
frames_per_buffer=フレーム数
```

というキーワード引数を与える点である。ストリームオブジェクトからサウンドを入力するには `read` メソッドを使用する。

《 サウンドの入力 》

書き方： ストリームオブジェクト.`read`(フレーム数)

これを実行することで、引数に与えたフレーム数のサウンドを入力してバイナリデータとして返す。

素朴なブロックモードによってサウンドを入力するプログラム `pyaudio03.py` を次に示す。

プログラム： `pyaudio03.py`

```
1  # coding: utf-8
2  import wave
3  import pyaudio
4  #####
5  # PyAudioによる 音声入力                                     #
6  #####
7  # PyAudioオブジェクトの生成
8  p = pyaudio.PyAudio()
9  # ストリームの生成
10 stm = p.open( format=pyaudio.paInt16,
11               channels=2, rate=44100,
12               frames_per_buffer=1024, input=True )
13 #-----
14 # ストリームからサウンドを取り込む回数 の 算定
15 # ・1 回の読み込みで1024フレームの入力
```

²⁵⁹入力デバイスの用意ができていないことが前提となる。この準備がなければ作成したプログラムは正しく動作しない。(エラーとなる)

```

16 # ・毎秒44100回のサンプリング
17 # ・5秒間のサウンド採取
18 # ということはデータの読み込み回数は次の通り
19 n = int( 44100 / 1024 * 5 )
20
21 # 5秒間の入力データをリストに蓄積
22 print('Recording started!')
23 dlist = [] # データリストの初期化
24 for i in range(n):
25     buf = stm.read(1024)
26     dlist.append(buf)
27 print('finished.')
28 # リストの要素を連結してバイナリデータ変換
29 data = b''
30 for i in range(n):
31     data += dlist[i]
32 print('data size: ', len(data))
33
34 # ストリームの終了処理
35 stm.stop_stream()
36 stm.close()
37 # PyAudioオブジェクトの廃棄
38 p.terminate()
39
40 #####
41 # WAV形式サウンドファイルへの書き込み #
42 #####
43 # ファイルのオープン
44 wf = wave.open('out02.wav', 'wb')
45 # 属性情報の取得
46 wf.setnchannels(2) # チャンネル数
47 wf.setsampwidth(2) # 量子化ビット数 (バイト数)
48 wf.setframerate(44100) # サンプリング周波数
49
50 # 内容の書き込み
51 wf.writeframes(data)
52 # ファイルのクローズ
53 wf.close()

```

解説

これは5秒間のサウンド入力を受け付け、それをWAV形式ファイルとして保存するものである。23～26行目の部分で入力したサウンドデータ（バイト列）を要素とするリスト `dlist` を作成し、29～31行目の部分でそのリストの要素を全て連結したバイトデータ `data` を作成している。

44行目以降の部分では、バイトデータ `data` をWAV形式ファイルとして保存している。

次にサウンド入力を別のスレッドで実行する方法について説明する。

コールバックモードでサウンドを入力すると、入力処理が別のスレッドで実行されるので、メインルーチンの処理の流れがブロックされない。コールバックモードのサウンド入力を行うプログラム `pyaudio04.py` を次に示す。

プログラム：pyaudio04.py

```

1 # coding: utf-8
2 import wave
3 import pyaudio
4
5 #####
6 # 書き込み用WAV形式サウンドファイルの準備 #
7 #####
8 # ファイルのオープン
9 wf = wave.open('out03.wav', 'wb')
10 # 属性情報の取得
11 wf.setnchannels(2) # チャンネル数
12 wf.setsampwidth(2) # 量子化ビット数 (バイト数)
13 wf.setframerate(44100) # サンプリング周波数
14
15 #####
16 # PyAudioによる音声入力 #
17 #####
18 #---- 入力用コールバック関数 (ここから) -----

```

```

19 def sndrec(in_data, frame_count, time_info, status):
20     wf.writeframes(in_data)
21     return ( None, pyaudio.paContinue )
22 #---- 入力用コールバック関数 (ここまで) -----
23
24 # PyAudioオブジェクトの生成
25 p = pyaudio.PyAudio()
26 # ストリームの生成
27 stm = p.open( format=pyaudio.paInt16, channels=2,
28               rate=44100, frames_per_buffer=1024,
29               input=True, stream_callback=sndrec )
30
31 # コールバック関数による入力を開始
32 stm.start_stream()
33
34 # 待ちループ: 「exit」と入力すれば終了処理に進む
35 while stm.is_active():
36     cmd = input('終了->exit: ')
37     if cmd == 'exit':
38         print('終了します...')
39         break
40
41 # ストリームの終了処理
42 stm.stop_stream()
43 stm.close()
44 # PyAudioオブジェクトの廃棄
45 p.terminate()
46
47 # WAV形式ファイルのクローズ
48 wf.close()

```

解説

コールバック関数の基本的な書き方は先に説明した通りであるが、このプログラムでは関数（sndrec）内の処理がWAV形式ファイルへの出力となっている。また return するタプルの最初の要素も今回は意味を持たないので None としている。

このプログラムではサウンド入力と同時に WAV 形式ファイルに保存しているが、フレームデータを累積するような処理にするなど、さまざまな応用が可能である。

付録

A Pythonに関する情報

A.1 Python のインターネットサイト

- 英語サイト：<https://www.python.org/>
- 日本語サイト：<https://www.python.jp/>
- Anaconda 公式サイト：<https://www.anaconda.com/>
- Anaconda 日本語情報：<https://www.python.jp/install/anaconda/>

A.2 Python のインストール作業の例

Python3 のインストール方法の例を挙げる。

A.2.1 PSF 版インストールパッケージによる方法

PSF 版インストールパッケージは、Python の公式団体（The Python Software Foundation）が配布する「純正の Python」とも呼べるディストリビューションである。PSF 版と後述の Anaconda では Python 環境の管理方法が異なるため、同一の計算機環境で両者を混在させる場合は注意を要する。²⁶⁰

【Windows にインストールする作業の例】

先に挙げた Python のインターネットサイトから Python のインストールプログラム（インストーラ：図 43）を入手する。



図 43: Windows 用インストーラ（64 ビット用）

インストーラのアイコンをダブルクリックしてインストールを実行する。図 44 は「Customize installation」を選んで詳細の設定を選択し、インストール対象のユーザは「all users」とした例である。各種ライブラリを導入、保守するためのツールである「pip」のインストールを指定している。

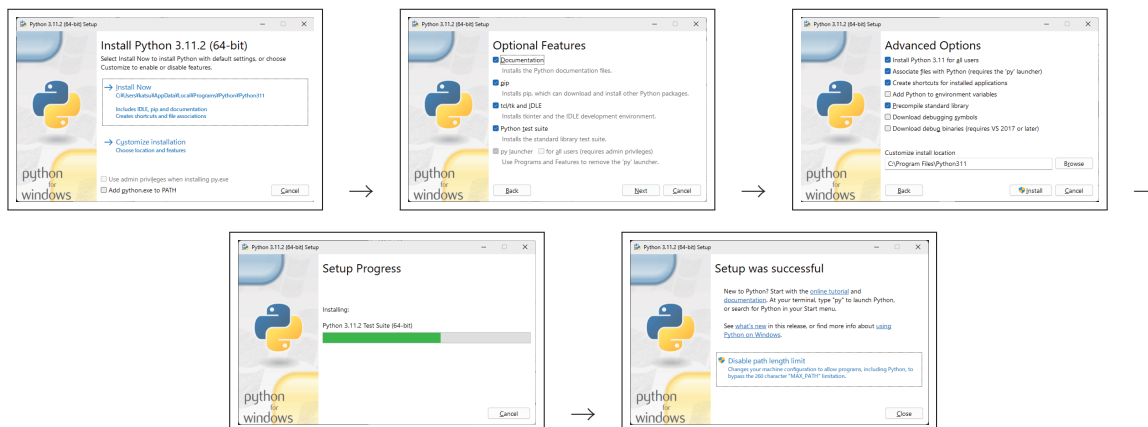


図 44: インストール処理の例（Windows）

²⁶⁰複数の異なる版の Python を同一計算機環境で扱うための仮想環境を構築するなどして安全に運用することも可能である。これに関しては本書では扱わない。

【Mac にインストールする作業の例】

Apple 社の macOS には既に Python2.7 がインストールされている²⁶¹ ことがあるが、Python3 を使用するには、先に挙げた Python のインターネットサイトから Python3 のインストーラ（図 45）を入手してインストールする。



図 45: Mac 用インストーラ

インストーラのアイコンをダブルクリックしてインストールを実行する。作業の流れの例を図 46 に示す。

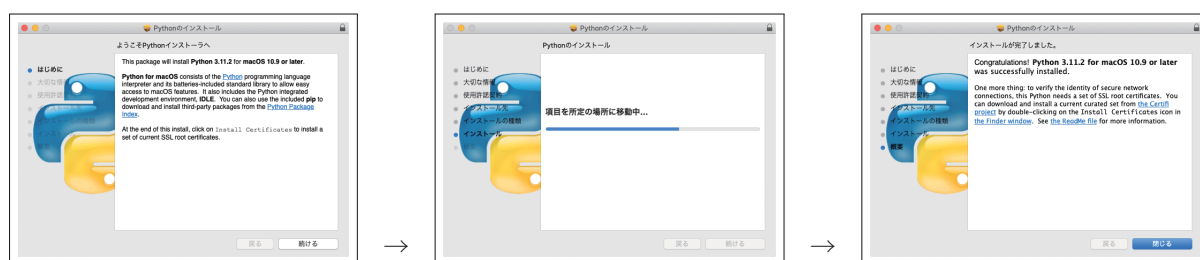


図 46: インストール処理の例（Mac）

A.2.2 Anaconda による方法

Anaconda はデータ処理の分野で Python を利用する際によく用いられるディストリビューションであり、多くのライブラリが予め同梱されている。各種 OS 用のものが配布されており、Python 環境全体の管理も比較的に簡単であることから、このディストリビューションの評価が高まってきている。

Anaconda は前述の PSF 版とパッケージの管理方法が異なるため、同一の計算機環境で両者を混在させる場合は注意を要する。（混在は推奨されない）

Anaconda のインストーラ（図 47）は前述の Anaconda 公式サイトから入手することができる。



Anaconda3-2022.10-Windows-x86_64.exe

図 47: Anaconda インストーラ

このインストーラを起動して、案内に沿ってインストール作業（図 48）を行う。

Anaconda は現在も活発に版が更新されており、インストール手順などに関しても公式サイトを参照されたい。

²⁶¹ macOS Monterey 12.3 の版から廃止された。

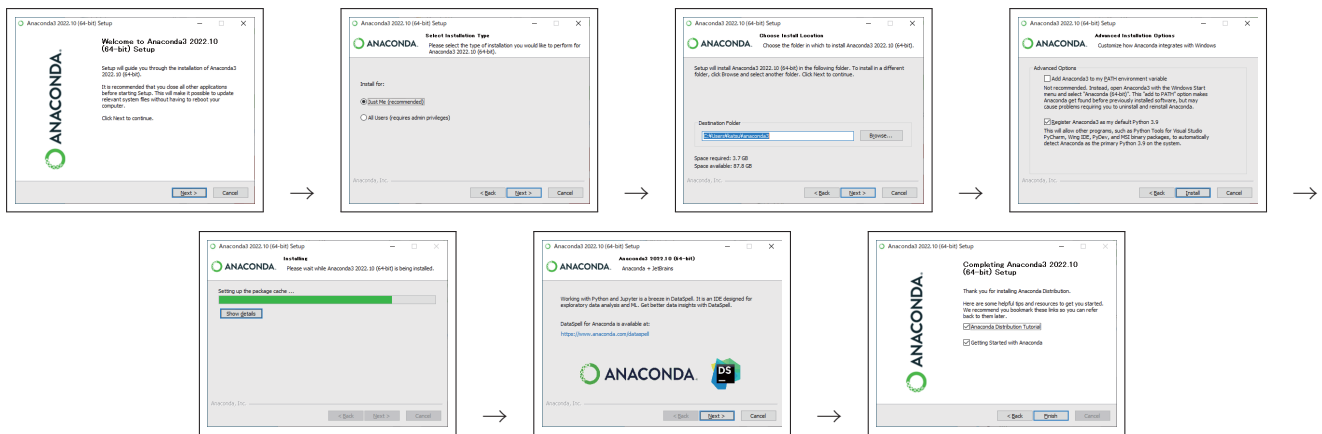


図 48: Anaconda のインストール作業の例

A.3 Python 起動のしくみ

A.3.1 PSF 版 Python の起動

PSF 版の Python 処理系は通常、**Python ランチャー** (Python Launcher) を介して起動される。標準的なインストールを行うと、Windows 環境では Python 本体は `C:\Program Files\Python311\python.exe` (Python3.11 の場合) としてインストールされる。もちろん、このパスを直接指定して Python 処理系を起動することもできるが、通常、Windows 環境では `py` コマンドを用いて起動する。この `py` コマンドが Windows 環境における **Python ランチャー** であり、このコマンドを用いることにより、複数の異なるバージョンの Python の起動を選択することができる。例えば 3.10 と 3.11 という 2 つの異なるバージョンの Python を 1 つの環境にインストールすることができ、次のようにして起動するバージョンを選択することができる。

```
py -3.10      :   Python3.10 を起動
py -3.11      :   Python3.11 を起動
```

Apple 社の macOS にも当該 OS 用の Python ランチャーがインストールされる。ただし、Windows の場合とは起動方法が異なる。Mac のコマンドでは、下記のように起動するバージョンを選択することができる。

```
python3.10    :   Python3.10 を起動
python3.11    :   Python3.11 を起動
```

Windows 環境の Python ランチャーでは、`-0` (ゼロ) オプションを指定することで、インストールされている Python のバージョンを確認することができる。(次の例参照)

例. インストールされている Python のバージョンの確認 (Windows 環境)

```
C:\Users\katsu>py -0 [Enter]          ←ゼロオプションによるバージョンの確認
-V:3.11 *      Python 3.11 (64-bit)    ←このバージョンがデフォルトで起動する
-V:3.10        Python 3.10 (64-bit)
```

この例では、3.10、3.11 (それぞれ 64 ビット版) がインストールされていることを示している。

A.3.2 Anaconda Navigator の起動

Anaconda では Anaconda Navigator (図 49) というインターフェースを用いて Python と関連ツールを利用する。

ENVIRONMENTS のタブ (図 50) で Anaconda のパッケージ管理ができる²⁶²。

A.3.3 Anaconda Prompt の起動

Anaconda の環境下でコマンド操作を行うには **Anaconda Prompt** を起動する。Windows 環境では Anaconda に関するツールをスタートメニュー (図 51) から起動する。

²⁶² パッケージ管理は `conda` コマンドでも可能である。特に Anaconda Navigator によるパッケージ管理が正しく動作しない場合などに `conda` コマンドが役立つ。`conda` コマンドに関しては後の「A.3.3.1 `conda` コマンドによる Python 環境の管理」(p.404) で解説する。

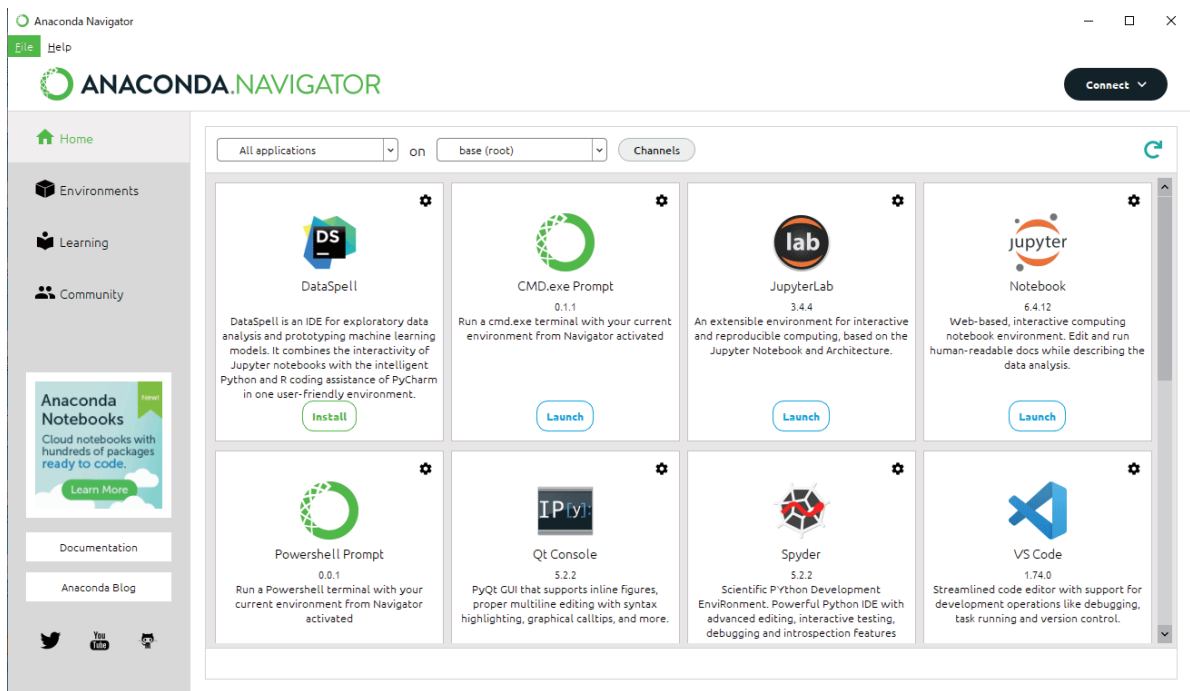


図 49: Anaconda Navigator

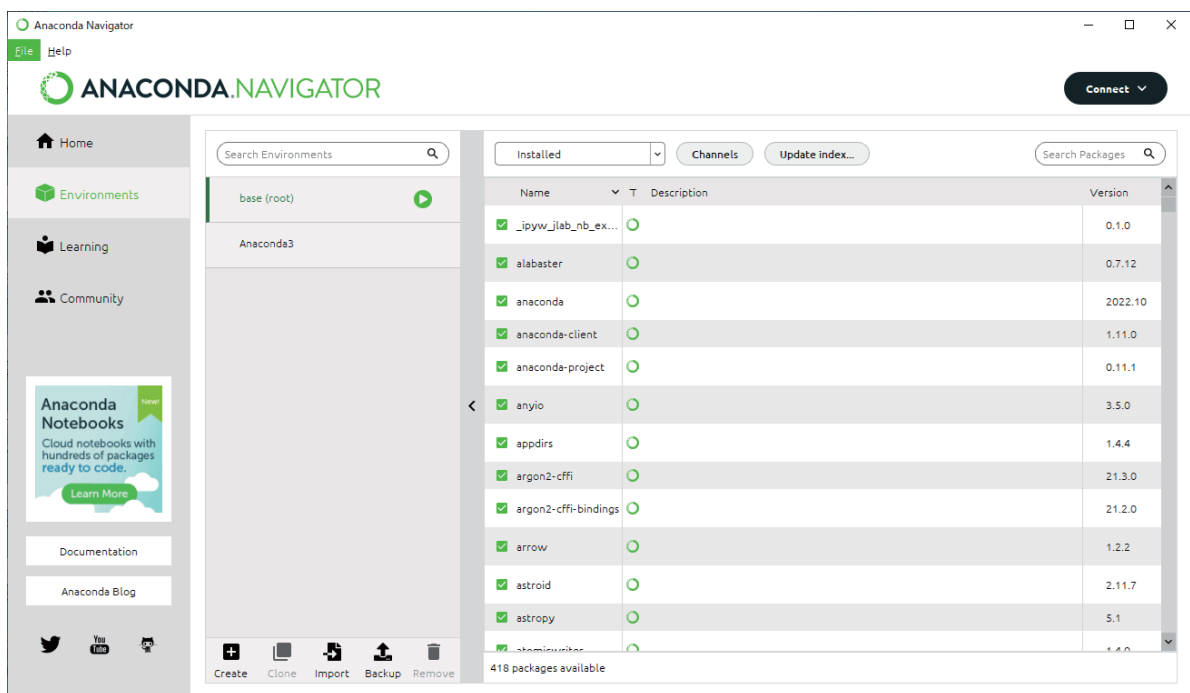


図 50: Anaconda Navigator でのパッケージ管理

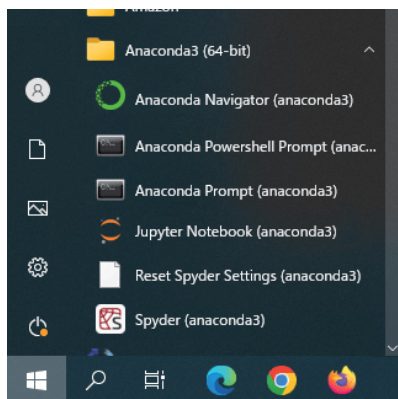


図 51: Windows のスタートメニューから使用する Anaconda

Anaconda 環境下でコマンドウィンドウを起動するには、図 51 中の「Anaconda Prompt」を選択する。

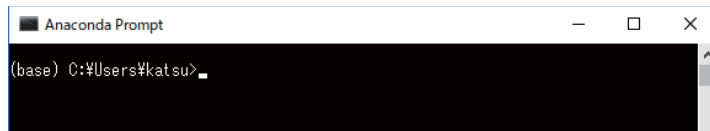


図 52: Anaconda Prompt を起動したところ

本書で例示する Python の対話モードと同様の操作を行うには、Anaconda Prompt から Python を起動するのが良い。(次の例参照)

例. Anaconda Prompt から Python を起動

```
(base) C:\Users\katsu> python [Enter]    ← Python インタプリタの起動
Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>    ←対話モードのプロンプト
```

これは python コマンドでインタプリタを起動している例である。

A.3.3.1 conda コマンドによる Python 環境の管理

Anaconda Prompt で conda コマンドによる Python 環境の管理ができる。(表 50 参照)

表 50: conda コマンドによる Python 環境の管理 (代表的な例)

コマンド	解説
conda update conda	Anaconda 環境のアップデート
conda update --all	Anaconda 環境下のソフトウェアのアップデート
conda update ライブラリ名	指定したライブラリのアップデート
conda list	インストールされているライブラリを一覧表示
conda search ライブラリ名	ライブラリをリポジトリ内で検索 ライブラリ名を省略すると入手可能なものの一覧が得られる。
conda install ライブラリ名	ライブラリのインストール
conda uninstall ライブラリ名	ライブラリのアンインストール

詳しくは Anaconda の公式情報サイトを参照のこと。

A.4 PIP によるライブラリ管理

PSF 版 Python では PIP を用いて各種のライブラリの導入と更新といった管理作業ができる。

▲注意▲ Anaconda の環境では PIP の使用は慎重にすること。 PSF 版 Python と Anaconda ではパッケージ管理の方法が異なる。

PIP は OS のコマンドシェル (Windows の場合はコマンドプロンプト) で pip コマンドを発行²⁶³ することで実行する。pip コマンド実行時の引数やオプションの代表的なものを表 51 に挙げる。

表 51: pip コマンド起動時の引数とオプション (一部)

実行方法	動作
pip list	現在インストールされているライブラリの一覧を表示する。
pip list -o	上記の内、より新しい版が入手可能なものを表示する。
pip install ライブラリ名	指定したライブラリを新規にインストールする。
pip install -U ライブラリ名	指定したライブラリを更新する。
pip uninstall ライブラリ名	指定したライブラリを削除する。

²⁶³Apple 社の OS X, macOS では pip3 コマンドで PIP を起動しなければならない場合がある。

例. requests ライブラリをインストールする作業

```
C:\Users\katsu> pip install requests  ←インストールの開始
Collecting requests
  Using cached requests-2.28.2-py3-none-any.whl (62 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in
  c:\program files\python311\lib\site-packages (from requests) (3.0.1)
Requirement already satisfied: idna<4,>=2.5 in
  c:\program files\python311\lib\site-packages (from requests) (3.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
  c:\program files\python311\lib\site-packages (from requests) (1.26.14)
Requirement already satisfied: certifi>=2017.4.17 in
  c:\program files\python311\lib\site-packages (from requests) (2022.12.7)
Installing collected packages: requests
Successfully installed requests-2.28.2
```

PIP でインストールできるライブラリは wheel という形式で配布されるものであり、表 51 の処理はインターネットに公開されている wheel ライブラリの情報に基いて実行される。従って PIP はインターネットに接続された計算機環境で使うことが前提となっているが、wheel 形式ライブラリのファイルをダウンロードして、オフライン環境で PIP を実行してそれをインストールすることも可能である。

wheel 形式ライブラリはファイル名として whl という拡張子を持ち、それをインストールするには

```
pip install wheel ライブラリ名.whl
```

というコマンド操作を行う。

例. ダウンロードした SciPy ライブラリをオフラインでインストールする作業

```
D:\work> pip install SciPy-1.8.1-cp311-cp311-win_amd64.whl  ←インストール開始
Processing d:\work\scipy-1.8.1-cp311-cp311-win_amd64.whl
...
(途中省略)
...
Installing collected packages: numpy, SciPy
Successfully installed SciPy-1.8.1 numpy-1.24.2
```

これは SciPy ライブラリのファイル SciPy-1.8.1-cp311-cp311-win_amd64.whl をインストールした例である。

A.4.1 PIP コマンドが実行できない場合の解決策

コマンドラインから pip が実行できない場合があるが、原因としては、コマンドサーチパスに pip コマンドのディレクトリが登録されていないことなどが挙げられる。その場合は、pip コマンドのディレクトリをコマンドサーチパスに登録すると解決するが、より簡単な解決策もある。それは python 処理系から pip を起動する方法である。具体的には、pip をコマンドとして起動するのではなく、次のようなコマンドラインとして投入する。

【Python を介して PIP を起動する方法】

Windows の場合： `py -m pip 引数の列...`

Mac の場合： `python3 -m pip 引数の列...`

例. インストールされているライブラリの一覧を表示する

```
py -m pip list
```

これを OS のコマンドラインから投入する。

B Kivyに関する情報

■ 公式インターネットサイト（英語）：<https://kivy.org/>

このサイトでインストール手順が説明されている。

B.1 Kivy 利用時のトラブルを回避するための情報

B.1.1 Kivy が使用する描画 API の設定

Kivy はグラフィックス描画の基礎に OpenGL ²⁶⁴ を用いるが、使用する計算機環境によっては、OS が提供する OpenGL の版が Kivy に適合しない場合もある。そのような場合は Kivy を使用したアプリケーションプログラムを実行する際にエラーや例外が発生する。しかし、Kivy では使用するグラフィックス用の API を環境変数の設定により選択することができるので、この件に関する問題が発生する場合は os モジュールの environ プロパティに使用する API を設定すると良い。具体的には次のような記述となる。

```
import os
os.environ['KIVY_GL_BACKEND'] = 'angle_sdl2'
```

指定できる API の例を表 52 に挙げる。

表 52: Kivy で使用できるグラフィックス API

指定する記号	解説
gl	UNIX 系 OS 用の OpenGL
glew（デフォルト）	Windows 環境で通常の場合に用いられる API
sdl2	Windows や UNIX 系 OS で gl や glew が使用できない場合にこれを用いる。 kivy.deps.sdl2 をインストールしておく必要がある。
angle_sdl2	Windows 環境で Python 3.5 以上の版を使用する際に利用できる。 kivy.deps.sdl2 と kivy.deps.angle をインストールしておく必要がある。

B.1.2 SDL について

Kivy は SDL（Simple DirectMedia Layer）を利用しているため、適切な版の SDL が計算機環境にインストールされている必要がある。SDL は公式インターネットサイト（<http://www.libsdl.org/>）から入手することができる。Kivy が動作しない場合は SDL の再インストールを検討するべきである。

B.2 GUI デザインツール

Kivy のための GUI アプリケーションデザインツール Kivy Designer が現在開発中であり、インターネットサイト <https://github.com/kivy/kivy-designer> から情報が入手できる。

²⁶⁴ グラフィックス描画のための、クロスプラットフォームの API。

C Tkinter：基本的な GUI ツールキット

Tkinter は、Python 処理系に標準的に添付されている GUI ツールキットであり、

```
import tkinter
```

とすることで必要なライブラリを読み込むことができる。このライブラリは Tcl/Tk ²⁶⁵ の Tk の部分を Python から利用するためのものであり、本書で取り上げた Kivy に比べて GUI のデザイン性と機能面が素朴である反面、その扱いが非常に容易である。本書では Tkinter の最も基本的な部分に限って使用例を示す。より詳しい情報に関しては Python の公式インターネットサイトなど²⁶⁶ を参照のこと。

C.1 基本的な扱い方

Tkinter では、GUI の要素を**ウィジェット**と呼び、様々なウィジェットを階層的に構成することで GUI アプリケーションを実現する。また、GUI アプリケーションの最上位のウィジェットに対してイベントループを実行する形で GUI アプリケーションの動作を実現する。

■ 最上位ウィジェットの生成

GUI アプリケーションの最上位のウィジェットのインスタンスは、Tk() を実行することで生成する。また、それに対して mainloop メソッドを実行することでイベントループが開始する。

Tk を用いた最も単純な GUI アプリケーションの実現方法を次に例示する。

例. 最も単純な（基本的な）GUI アプリケーション

```
>>> import tkinter      Enter      ← Tkinter ライブラリの読み込み
>>> tkinter.Tk().mainloop()      Enter      ←最上位ウィジェットの生成とイベントループの開始
```

これにより図 53 のような GUI アプリケーションのウィンドウが表示される。

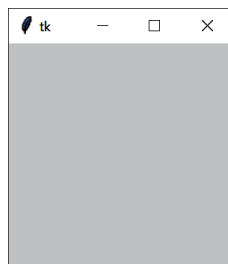


図 53: Tkinter による最も簡単な GUI アプリケーション

この例では、tkinter.Tk() で GUI の最上位のウィジェットを生成し、それに対して mainloop() を実行している。これによりイベントを待ち受けるループが起動し、ウィンドウを閉じることでアプリケーションが終了する。

実用的な GUI アプリケーションを構築する流れは次のようなものである。

【GUI アプリケーションの構築と実行の流れ】

1. 最上位ウィジェットの生成

これが GUI アプリケーションとなる。

2. 各種ウィジェットを生成して上記ウィジェット配下に登録する

Frame オブジェクトを生成して、それを GUI のコンテナとして使用し、それに対してラベル、ボタン、入力フィールドなど、各種のウィジェットを生成して配置する。各種ウィジェットにはイベントに対する処理のための関数を登録する。（イベントハンドリングの実装）

ここで生成した Frame オブジェクトを最上位のウィジェットに登録する。

²⁶⁵Tcl/Tk: GUI ツールキット Tk をスクリプト言語 Tcl から利用することで GUI 形式のアプリケーションプログラムを実現するスクリプティング環境である。Tk は Tcl 以外の各種の言語からも API として呼び出すことで利用することができる。

²⁶⁶ニューメキシコ工科大学 (<http://www.nmt.edu/>) が編纂し公開している “Tkinter8.5 reference: a GUI for Python” (John W. Shipman 著) が Tkinter に関して詳しい。Web 版と PDF 版が公開されている。

3. イベントループの開始

最上位のウィジェットに対して `mainloop()` を実行して GUI アプリケーションを開始する。(イベントループの開始)

使用例を次に示す。

C.1.1 使用例

ここでは、ボタンに反応する GUI アプリケーション `tk01.py` の実装例を示す。このアプリケーションは図 54 に示すように、「表示」「消去」の2つのボタンを備え、「表示」をクリックするとメッセージを表示し、「消去」をクリックするとそのメッセージを消去する。

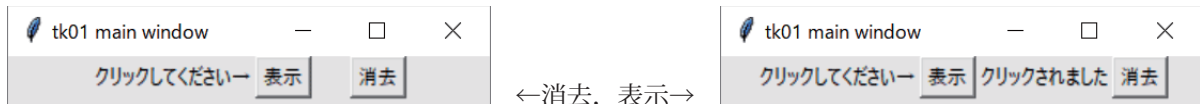


図 54: ボタンに反応する GUI アプリケーション

実装したプログラムの例 `tk01.py` を次に示す。

プログラム：tk01.py

```
1  # coding: utf-8
2  import tkinter      # Tkinterの読み込み
3
4  #--- イベントハンドリング用の関数 ---
5  def cmdDisp():      # 「表示」ボタンに対する処理
6      global lb2
7      lb2['text'] = 'クリックされました'
8      print('「表示」ボタンがクリックされました')
9
10 def cmdDel():       # 「消去」ボタンに対する処理
11     global lb2
12     lb2['text'] = ' '
13     print('「消去」ボタンがクリックされました')
14
15 #--- GUIの構築 ---
16 root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
17 root.title('tk01 main window') # ウィンドウのタイトル
18 root.geometry('300x30+20+10') # ウィンドウのサイズと位置
19
20 fr = tkinter.Frame(root) # GUI構築用のコンテナ
21 fr.pack()                # frをrootに配置
22
23 lb1 = tkinter.Label(fr, text='クリックしてください→') # 文字のラベル1
24 lb1.pack(side='left') # lb1をfrに配置
25
26 b1 = tkinter.Button(fr, text='表示', command=cmdDisp) # ボタン1
27 b1.pack(side='left') # b1をfrに配置
28
29 lb2 = tkinter.Label(fr, text=' ') # 文字のラベル2 (文字なし)
30 lb2.pack(side='left') # lb2をfrに配置
31
32 b2 = tkinter.Button(fr, text='消去', command=cmdDel) # ボタン2
33 b2.pack(side='left') # b2をfrに配置
34
35 #--- アプリケーションの起動 ---
36 root.mainloop()        # イベントループ
```

解説：

「表示」ボタンのクリックに対して実行する関数 `cmdDisp` と、「消去」ボタンのクリックに対して実行する関数 `cmdDel` が5～13行目に定義されている。

16行目で、アプリケーションの最上位のウィジェット `root` を生成し、17行目でウィンドウのタイトルを設定 (`title` メソッド) し、18行目でウィンドウのサイズと位置を設定 (`geometry` メソッド) している。

GUIを構築するためのコンテナとして、20行目でFrameオブジェクト `fr` を生成している。このときコンストラクタの第1引数に、登録先の上位のオブジェクトを与える。更に21行目で、`pack` メソッドを用いて、配置（表示位置）を決定している。

文字を表示するためのラベルウィジェットは `Label` クラスのオブジェクトであり、コンストラクタの第1引数に「親ウィジェット」（登録先の上位ウィジェット）を、キーワード引数 `'text='` に表示する文字列を与える。23行目で `Label` ウィジェットを生成しており、24行目で上位の `fr` の上での配置を設定している。29～30行目でも同様の手順で `Label` オブジェクトを生成・配置している。ラベルウィジェット生成後に表示内容（テキスト）を与えるには、そのラベルオブジェクトにスライス `['text']` を付けたものに値を設定する。（7,12行目）

ボタンウィジェットは `Button` クラスのオブジェクトであり、コンストラクタの第1引数に「親ウィジェット」（登録先の上位ウィジェット）を、キーワード引数 `'text='` に表示する文字列を与える。また、キーワード引数 `'command='` にクリックされた際に実行する関数の名前を与える。26行目で「表示」の `Button` ウィジェットを生成しており、27行目で上位の `fr` 上での配置を設定している。32～33行目でも同様の手順で `Button` オブジェクトを生成・配置している。

C.1.1.1 ウィンドウサイズ変更の可否の設定

`Tk()` で生成した最上位ウィジェットに対して `resizable` メソッドを使用することで、ウィンドウのサイズ変更の可否を設定できる。

書き方： `ウィジェット.resizable(横方向の変更可否, 縦方向の変更可否)`

このメソッドの引数には真理値（`True/False`）を与える。`False` を与えるとリサイズの禁止が設定される。

C.1.2 ウィジェットの配置

先の `tk01.py` の例では、`pack` メソッドを使用してウィジェットを配置した。`pack` メソッドはウィジェットを「縦方向もしくは横方向の一次元」に配置するものであるが、この他にも二次元的な配置や、座標指定による自由な配置のためのメソッドもある。

【サンプルプログラム】

ウィジェットの配置を確認するためのサンプルプログラム `tk02.py` を次に示す。このプログラムは、`pack` メソッドによる一次元の配置と、`grid` メソッドによる二次元の配置を確認するためのものである。後に示す「試み」により確認できる。

プログラム： `tk02.py`

```
1  # coding: utf-8
2  import tkinter          # Tkinterの読み込み
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
6  root.title('tk02 main window') # ウィンドウのタイトル
7  root.geometry('300x60+20+10') # ウィンドウのサイズと位置
8
9  fr = tkinter.Frame(root) # GUI構築用のコンテナ
10 fr.pack()                # frをrootに配置
11
12 # 配置テスト用のボタン群
13 b11 = tkinter.Button(fr, text='b11')
14 b12 = tkinter.Button(fr, text='b12')
15 b21 = tkinter.Button(fr, text='b21')
16 b22 = tkinter.Button(fr, text='b22')
17
18 #--- 配置 ---
19 # 横方向
20 b11.pack(side='left');    b12.pack(side='left')
21 b11.pack(side='right');   b12.pack(side='right')
22 # 縦方向
23 b11.pack(side='top');     b21.pack(side='top')
24 b11.pack(side='bottom');  b21.pack(side='bottom')
25
26 # 縦横（二次元）
```

```

27 #b11.grid(row=0,column=0); b12.grid(row=0,column=1)
28 #b21.grid(row=1,column=0); b22.grid(row=1,column=1)
29
30 #--- アプリケーションの起動 ---
31 root.mainloop() # イベントループ

```

試み：(1) 一次元の配置

- ・ 20 行目、21 行目のどちらかのコメント「#」を外すことで、pack メソッドによる横方向の配置が確認できる。
- ・ 23 行目、24 行目のどちらかのコメント「#」を外すことで、pack メソッドによる縦方向の配置が確認できる。

pack メソッドに与えるキーワード引数 'side=' の値によって配置がどのようなになるかを確認すること。

試み：(2) 二次元の配置

- ・ 27 行目、28 行目の両方のコメント「#」を外すことで、grid メソッドによる縦横の配置が確認できる。

grid メソッドに与えるキーワード引数 'row=','column=' の値によって行と列の位置が指定できる。

tk02.py の実行結果の例を図 55 に示す。

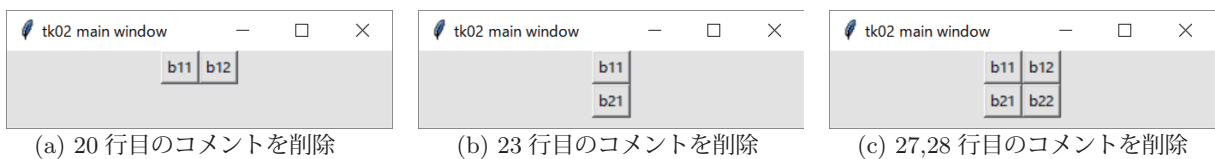


図 55: pack や grid メソッドによるボタンの配置

コメントの削除によってボタンの配置の様子が確認できる。

■ ウィジェットの伸縮の制御

pack でウィジェットを配置する際、キーワード引数 'fill=','expand=' に値を設定（表 53）することにより、ウィジェットの伸縮を制御できる。

表 53: キーワード引数 'fill=' と 'expand='

キーワード引数	値	効果
fill=	'x'	横方向に伸ばす
	'y'	縦方向に伸ばす
	'both'	縦横両方向に伸ばす
expand=	True	サイズの拡張を可能にする
	False	サイズを拡張しない

ラベルウィジェットの伸縮制御を試みるサンプルプログラム tk02-2.py を示す。

プログラム：tk02-2.py

```

1  # coding: utf-8
2  import tkinter # Tkinterの読み込み
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk() # アプリケーションの最上位ウィジェット
6  root.title( 'tk02-2 main window' ) # ウィンドウのタイトル
7  root.geometry( '280x280+20+10' ) # ウィンドウのサイズと位置
8
9  fr = tkinter.Frame(root) # GUI構築用のコンテナ
10 fr.pack( fill='both', expand=True ) # frをrootに配置
11
12 # 配置テスト用のラベル
13 lb1 = tkinter.Label( fr, text='ラベル 1', font='32',
14                      fg='#ff0000', bg='#ffff00' )
15 lb1.pack( fill='x', expand=True ) # 横に伸ばす
16
17 lb2 = tkinter.Label( fr, text='ラベル 2', font='32',
18                      fg='#ffff00', bg='#ff0000' )
19 lb2.pack( fill='both', expand=True ) # 縦横に伸ばす

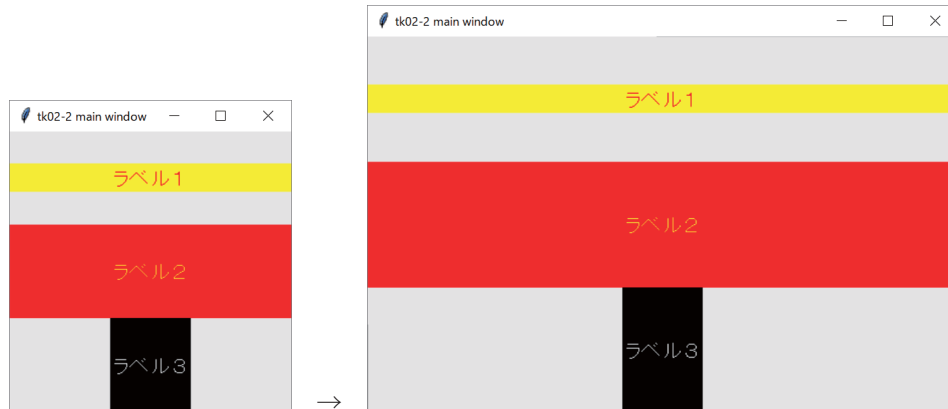
```

```

20
21 lb3 = tkinter.Label( fr, text='ラベル 3', font='32',
22                      fg='#ffffff', bg='#000000' )
23 lb3.pack( fill='y', expand=True )      # 縦に伸ばす
24
25 #--- アプリケーションの起動 ---
26 root.mainloop()                      # イベントループ

```

このプログラムの実行例を図 56 に示す。



ウィンドウサイズの変更に伴ってラベルのサイズが変わる

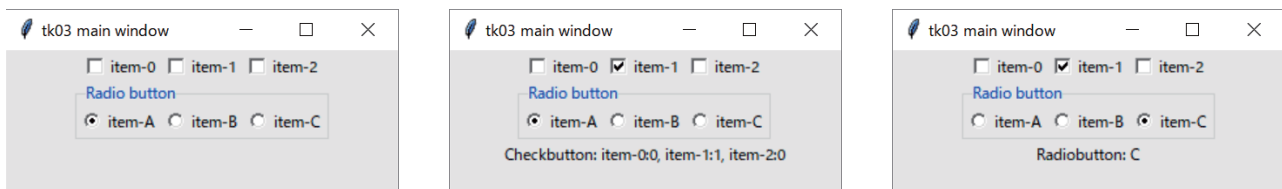
図 56: ウィジェットの伸縮

C.2 各種のウィジェット

ここでは、特に利用頻度の高いウィジェットを紹介し、その使用方法を例示する。

C.2.1 チェックボタンとラジオボタン

チェックボタン²⁶⁷ とラジオボタンの使用方法についてサンプルプログラムを示しながら説明する。ここでは図 57 のような GUI アプリケーションを考える。



(a) アプリケーション起動直後

(b) チェックボタンを操作した場合の表示

(c) ラジオボタンを操作した場合の表示

図 57: チェックボタンとラジオボタン

このアプリケーションは最上段に 3 つのチェックボタン、中段に 3 つのラジオボタン、最下段に文字列表示用のラベルを備えている。チェックボタンやラジオボタンを操作すると、それらの状態をラベルに報告する。

・チェックボタンのコンストラクタ

Checkbutton(上位ウィジェット, text=ボタンに表示する文字列,
onvalue=既チェック時の値, offvalue=未チェック時の値, variable=状態保持用オブジェクト,
command=操作時に起動する関数の名前)

チェックボタンは「既チェック／未チェック」の状態を保持するウィジェットであり、その状態を状態保持用オブジェクト (Variable クラスのオブジェクト) に保持する。

C.2.1.1 Variable クラス

Variable クラスは各種状態 (値) を保持するための次のような各種クラスを提供する。

²⁶⁷Tkinter 以外の GUI ツールキットでは「チェックボックス」と呼ぶことが多い。

StringVar	:	文字列の値を保持するオブジェクト
IntVar	:	整数の値を保持するオブジェクト
DoubleVar	:	浮動小数点数の値を保持するオブジェクト
BooleanVar	:	真理値を保持するオブジェクト

これらクラスのオブジェクトは、値の変更などをイベントとして検知する²⁶⁸ 機能があり、Tkinter の他の様々なウィジェットでも使用する。Variable オブジェクトに対して set メソッドや get メソッドを使用して値の設定と参照ができる。

・ラジオボタンのコンストラクタ

Radiobutton(上位ウィジェット, text=ボタンに表示する文字列,
value=選択時の値, variable=状態保持用オブジェクト, command=操作時に起動する関数の名前)

複数のラジオボタンを同一のグループでまとめるには「状態保持用オブジェクト」として同一のオブジェクトを指定する。また図 57 にもある通り、同一グループのラジオボタンは見出し付きの枠で囲んで表示することも多く、その場合は上位ウィジェットとして、ラベルフレーム (tkinter.ttk.Labelframe クラスのオブジェクト) を使用する。

図 57 のアプリケーションの実装例を tk03.py に示す。

プログラム：tk03.py

```

1  # coding: utf-8
2  import tkinter          # Tkinterの読み込み
3  from tkinter import ttk  # tkinter.ttkの読み込み
4
5  #--- イベントハンドリング用の関数 ---
6  def cmdChk():
7      msg = 'Checkbutton'+\
8          ': item-0:'+str(vChk1.get())+\
9          ', item-1:'+str(vChk2.get())+\
10         ', item-2:'+str(vChk3.get())
11     lbText['text'] = msg
12
13  def cmdRd():
14     msg = 'Radiobutton: '+vRd.get()
15     lbText['text'] = msg
16
17  #--- GUIの構築 ---
18  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
19  root.title('tk03 main window') # ウィンドウのタイトル
20  root.geometry('300x110+20+10') # ウィンドウのサイズと位置
21
22  frMain = tkinter.Frame(root) # GUI構築用のコンテナ
23  frMain.pack() # frMainをrootに配置
24
25  #-----
26  frChk = tkinter.Frame(frMain) # チェックボタン用コンテナ
27  frChk.pack( side='top' )      # frChkをfrMainに配置
28
29  # チェックボタン 1
30  vChk1 = tkinter.IntVar()      # Variableオブジェクト
31  btnChk1 = tkinter.Checkbutton( frChk, text='item-0',
32                                 onvalue=1, offvalue=0, variable=vChk1,
33                                 command=cmdChk )
34  btnChk1.pack( side='left' )
35
36  # チェックボタン 2
37  vChk2 = tkinter.IntVar()      # Variableオブジェクト
38  btnChk2 = tkinter.Checkbutton( frChk, text='item-1',
39                                 onvalue=1, offvalue=0, variable=vChk2,
40                                 command=cmdChk )
41  btnChk2.pack( side='left' )
42
43  # チェックボタン 3

```

²⁶⁸Variable クラスのオブジェクトに対して trace メソッドを用いてコールバック関数を登録できる。後述の「C.2.5 スケール (スライダ) とプログレスバー」(p.417) のところで事例を紹介する。

```

44 vChk3 = tkinter.IntVar()          # Variableオブジェクト
45 btnChk3 = tkinter.Checkbutton( frChk, text='item-2',
46                                onvalue=1, offvalue=0, variable=vChk3,
47                                command=cmdChk )
48 btnChk3.pack( side='left' )
49
50 #-----
51 # ラジオボタン用ラベルフレーム
52 lfrRd = ttk.Labelframe(frMain, text='Radio button')
53 lfrRd.pack( side='top' )
54
55 vRd = tkinter.StringVar()        # ラジオボタン用変数
56 vRd.set('A')
57
58 # ラジオボタン 1
59 btnRd1 = tkinter.Radiobutton( lfrRd, text='item-A',
60                               variable=vRd, value='A', command=cmdRd )
61 btnRd1.pack( side='left' )
62 # ラジオボタン 2
63 btnRd2 = tkinter.Radiobutton( lfrRd, text='item-B',
64                               variable=vRd, value='B', command=cmdRd )
65 btnRd2.pack( side='left' )
66 # ラジオボタン 3
67 btnRd3 = tkinter.Radiobutton( lfrRd, text='item-C',
68                               variable=vRd, value='C', command=cmdRd )
69 btnRd3.pack( side='left' )
70
71 #-----
72 # テキスト表示用ラベル
73 lbText = tkinter.Label(frMain)
74 lbText.pack( side='top' )
75
76 #--- アプリケーションの起動 ---
77 root.mainloop()                # イベントループ

```

C.2.2 エントリ（テキストボックス）とコンボボックス

文字列を入力するにはエントリあるいは、tkinter.ttk が提供するコンボボックスが利用できる。

・エントリのコンストラクタ

Entry(上位ウィジェット, textvariable=状態保持用オブジェクト, width=長さ)

エントリは基本的な文字入力用ウィジェットである。コンボボックスは文字の入力に加えて、予め設定しておいた選択肢から選択する形式の入力も可能である。

・コンボボックスのコンストラクタ

Combobox(上位ウィジェット, textvariable=状態保持用オブジェクト, width=長さ)

コンボボックス (tkinter.ttk.Combobox クラスのオブジェクト) に選択肢を設定するには、そのオブジェクトにスライス ['values'] を付けたものに選択肢を要素として持つタプルを与える。

キーワード引数 'width=' にはウィジェットの横幅を設定する。

【サンプルプログラム】

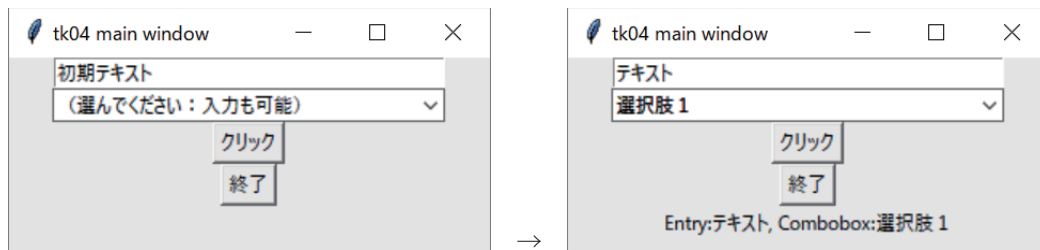
エントリとコンボボックスを使用した、図 58 のような動作をするサンプルプログラムを tk04.py に示す。

プログラム：tk04.py

```

1  # coding: utf-8
2  import tkinter                # Tkinter
3  from tkinter import ttk      # tkinter.ttk
4
5  #--- GUIの構築 ---
6  root = tkinter.Tk()          # アプリケーションの最上位ウィジェット
7  root.title( 'tk04 main window' ) # ウィンドウのタイトル
8  root.geometry( '300x120+20+10' ) # ウィンドウのサイズと位置
9
10 frMain = tkinter.Frame(root)  # GUI構築用のコンテナ

```



「クリック」ボタンにより入力内容がラベルに表示される。

図 58: チェックボタンとラジオボタン

```

11 frMain.pack()      # frMainをrootに配置
12
13 #-----
14 # エントリ
15 vEnt1 = tkinter.StringVar()      # エントリ用変数
16 vEnt1.set('初期テキスト')
17
18 ent1 = tkinter.Entry(frMain, width=40, textvariable=vEnt1)
19 ent1.pack( side='top' )
20
21 #-----
22 # コンボボックス
23 vCmb1 = tkinter.StringVar()      # コンボボックス用変数
24 vCmb1.set(' (選んでください：入力も可能) ')
25
26 cmb1 = ttk.Combobox(frMain, width=37, textvariable=vCmb1)
27 cmb1['values'] = ('選択肢 1','選択肢 2','選択肢 3')
28 cmb1.pack( side='top' )
29
30 #-----
31 # ボタン
32
33 # 内容表示ボタン
34 def cmdBtn1():      # ボタンのハンドリング用関数
35     global lb1
36     lb1['text'] = 'Entry:'+vEnt1.get()+', Combobox:'+vCmb1.get()
37
38 btn1 = tkinter.Button(frMain,text='クリック', command=cmdBtn1)
39 btn1.pack( side='top' )
40
41 # 終了ボタン
42 def cmdBtn2():      # ボタンのハンドリング用関数
43     print('終了します。')
44     root.quit()      # アプリケーションの終了
45
46 btn2 = tkinter.Button(frMain,text='終了', command=cmdBtn2)
47 btn2.pack( side='top' )
48
49 #-----
50 # ラベル
51 lb1 = tkinter.Label(frMain,width=40)
52 lb1.pack( side='top' )
53
54 #--- アプリケーションの起動 ---
55 root.mainloop()      # イベントループ

```

C.2.2.1 プログラムの終了

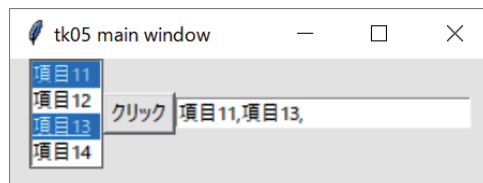
tk04.py の 44 行目にあるように、quit メソッドを実行するとアプリケーションが終了する。

C.2.3 リストボックス

複数の要素を持つリストの中から項目を選択するリストボックスを実現するための Listbox クラスについて説明する。ここでは図 59 に示すようなアプリケーションの作成を例に示す。



リスト項目を選択してボタンをクリックすると選択内容がエントリに表示される。



複数の項目を選択するリストボックスも実現可能

図 59: リストボックス

図 59 に示すアプリケーションは、左端のリストから項目を選択し、「クリック」ボタンをクリックすると右端のエントリに選択した項目を表示するものである。このアプリケーションの実装例を tk05.py に示す。

プログラム：tk05.py

```

1  # coding: utf-8
2  import tkinter                # Tkinter
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()           # アプリケーションの最上位ウィジェット
6  root.title( 'tk05 main window' ) # ウィンドウのタイトル
7  root.geometry( '300x80+20+10' ) # ウィンドウのサイズと位置
8
9  frMain = tkinter.Frame(root)   # GUI構築用のコンテナ
10 frMain.pack() # frMainをrootに配置
11
12 # リストボックス
13 vLbx1 = tkinter.StringVar( value=( '項目11', '項目12', '項目13', '項目14' ) )
14 lbx1 = tkinter.Listbox( frMain, listvariable=vLbx1, height=4, width=7,
15 #                          selectmode='single' # 選択：シングル
16                          selectmode='multiple' # 選択：複数
17 )
18 lbx1.pack( side='left' )
19
20 # ボタン
21 def cmdBtn1(): # クリックによって起動する関数
22     global vEnt1, lbx1
23     msg = ''
24     for i in lbx1.curselection():
25         msg += lbx1.get(i)+','
26     vEnt1.set( msg )
27
28 btn1 = tkinter.Button( frMain, text='クリック', command=cmdBtn1 )
29 btn1.pack( side='left' )
30
31 # エントリ
32 vEnt1 = tkinter.StringVar()
33 ent1 = tkinter.Entry( frMain, textvariable=vEnt1, width=30 )
34 ent1.pack( side='left' )
35
36 #--- アプリケーションの起動 ---
37 root.mainloop()                # イベントループ

```

・リストボックスのコンストラクタ

Listbox(上位ウィジェット, listvariable=状態保持用オブジェクト, width=長さ, height=高さ, selectmode=選択モード)

キーワード引数 'listvariable=' には StringVar オブジェクトを与えるが、それを生成する際のコンストラクタのキーワード引数 'value=' に、選択肢の文字列を要素とするタプルを与える。リストボックスには選択モードが設定でき、単

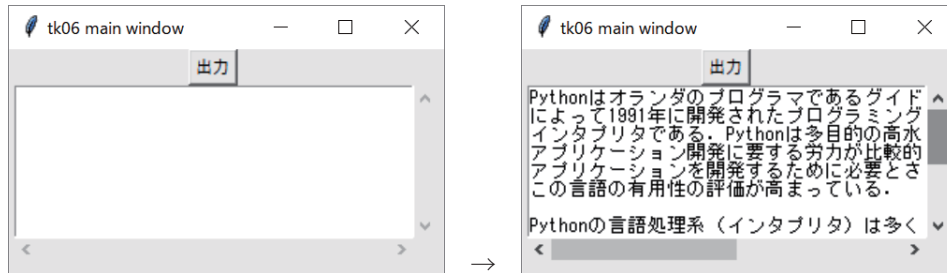
一の選択肢を選ぶか、複数の選択肢を同時に選ぶかを設定できる。これはキーワード引数 'selectmode=' に 'single' か 'multiple' を与えることで設定する。

リストボックスの選択されている項目を取得するには curselection メソッド (24 行目) を使用する。これにより、選択されている項目番号のタプルが得られる。リストボックスの n 番目の項目の値を取り出すには get(n) メソッドを実行する。(25 行目)

C.2.4 テキスト（文字編集領域）とスクロールバー

先に紹介したエントリは、1 行分の入力を想定しているが、テキストは複数行のテキストの編集を可能にするウィジェットである。ここではテキストに加えて、スクロールバーの設置についても説明する。

図 60 のようなアプリケーションの実装について考える。



テキストの入力に応じて、上下、左右のスクロールバーが連動する

図 60: スクロールバーと連動するテキスト（文字編集領域）

これは、ウィンドウの編集領域に文字を入力し、「出力」ボタンをクリックすると内容を標準出力に出力するものである。実装例を tk06.py に示す。

プログラム：tk06.py

```
1  # coding: utf-8
2  import tkinter          # Tkinter
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
6  root.title( 'tk06 main window' ) # ウィンドウのタイトル
7  root.geometry( '310x160+20+10' ) # ウィンドウのサイズと位置
8
9  frMain = tkinter.Frame(root) # GUI構築用のコンテナ
10 frMain.pack() # frMainをrootに配置
11
12 # ボタン
13 def cmdBtn1():
14     global txt
15     t = txt.get('1.0', 'end').rstrip()
16     print('--- top ---')
17     print(t)
18     print('--- bottom ---')
19
20 btn1 = tkinter.Button( frMain, text='出力', command=cmdBtn1 ) # ボタン
21 btn1.grid( row=0, column=0 )
22
23 # テキスト
24 txt = tkinter.Text( frMain, width=40, height=8, wrap='none' )
25 txt.grid( row=1, column=0 )
26
27 # スクロールバー（縦）
28 scrY = tkinter.Scrollbar( frMain, orient='vertical', command=txt.yview )
29 txt['yscrollcommand'] = scrY.set
30 scrY.grid( row=1, column=1, sticky=('n', 's') )
31
32 # スクロールバー（横）
33 scrX = tkinter.Scrollbar( frMain, orient='horizontal', command=txt.xview )
34 txt['xscrollcommand'] = scrX.set
35 scrX.grid( row=2, column=0, sticky=('w', 'e') )
36
```

```

37 #--- アプリケーションの起動 ---
38 root.mainloop()                # イベントループ

```

• テキストのコンストラクタ

Text(上位ウィジェット, width=横幅, height=高さ, wrap=折返し指定)

折返し指定に 'none' を与えると, Text の横幅よりも長い行の折り返し処理をしない。

• スクロールバーのコンストラクタ

Scrollbar(上位ウィジェット, orient=縦横の方向, command=関数名)

縦横の方向に 'vertical' を与えると縦, 'horizontal' を与えると横のスクロールバーとなる。キーワード引数 'command=' には今回は txt.yview や txt.xview を与えているが, これは Text オブジェクトの表示位置を変更するメソッドの名前である。(このメソッドに関して詳しいことは Tk のドキュメントを参照のこと) これにより, スクロールバーの位置の変更が, Text オブジェクトの txt の表示位置に反映される。

また, Text オブジェクト txt の内容編集に伴って表示位置の変更が発生した場合は, それをスクロールバーに反映させる必要があるが, それは Text オブジェクトにスライス ['yscrollcommand'], ['xscrollcommand'] を付けたものに, スクロールバーの位置変更を施すメソッドの名前 'set' を与えている。(set メソッドに関しても詳しくは Tk のドキュメントを参照のこと)

今回のプログラムでは, スクロールバーを grid メソッドで配置する際に, キーワード引数 'sticky=' を与えて上下あるいは左右いっぱいに広げている, (30,35 行目) 'sticky=' に与える値に関しては p.421 の図 64 「位置の基準」を参照のこと。このような設定により Text オブジェクト txt の縦横のサイズにスクロールバーのサイズを合わせている。

このアプリケーションの「出力」ボタンをクリックした際の標準出力の例を次に示す。

出力の例

— top —

Python はオランダのプログラマであるガイド・ヴァンロッサム (Guido van Rossum) によって 1991 年に開発されたプログラミング言語であり, 言語処理系は基本的にインタプリタである。Python は多目的の高水準言語であり, 言語そのものの習得とアプリケーション開発に要する労力が比較的に少ないとされる。しかし, 実用的なアプリケーションを開発するために必要とされる多くの機能が提供されており,

⋮
(途中省略)
⋮

ものの習得や運用の簡便性と相俟って, 情報工学や情報科学とは縁の遠い分野の利用者に対してもアプリケーション開発の敷居を下げている。

— bottom —

Text オブジェクトから内容を取り出すには get メソッドを使用する。その際, 第 1 引数にテキストの開始位置, 第 2 引数に終了位置を指定する。プログラムの 15 行目では開始位置として '1.0' を与えているが, これは '行番号. 桁' の形で指定する形式であり「第 1 行目のインデックス 0 番目」を意味する。

C.2.5 スケール (スライダ) とプログレスバー

スケールはいわゆる「スライダ」とも呼ばれるウィジェットで, 直感的な操作で 1 次元の数値を設定するものである。また, プログレスバーは 1 次元の数値を直感的に図示するウィジェットである。ここでは図 61 に示すようなアプリケーションの構築を例に挙げて, スケールとプログレスバーの扱いについて説明する。



図 61: スケールにプログレスバーを連動させる試み

このアプリケーションは、スケールで設定された値をそのままプログレスバーに反映させるものである。プログラムの例を tk07.py に示す。

プログラム：tk07.py

```
1  # coding: utf-8
2  import tkinter          # Tkinter
3  from tkinter import ttk  # tkinter.ttk
4
5  #--- GUIの構築 ---
6  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
7  root.title( 'tk07 main window' )  # ウィンドウのタイトル
8  root.geometry( '290x100+20+10' )  # ウィンドウのサイズと位置
9
10 frMain = tkinter.Frame(root)  # GUI構築用のコンテナ
11 frMain.pack()  # frMainをrootに配置
12
13 # スケール
14 def traceSc1( *arg ):          # 値の変更を受けて起動する関数
15     global vSc1, pb1
16     pb1.configure( value=vSc1.get() )  # プログレスバーの状態の変更
17
18 vSc1 = tkinter.DoubleVar()      # スケールの値を保持するオブジェクト
19 vSc1.set(0)                    # 初期値
20 vSc1.trace('w',traceSc1)        # 値が変更されたときのハンドリング
21
22 sc1 = tkinter.Scale( frMain, variable=vSc1 )  # スケールオブジェクト
23 sc1.pack( side='left' )
24
25 # プログレスバー
26 pb1 = ttk.Progressbar(frMain)
27 pb1.pack( side='left' )
28
29 #--- アプリケーションの起動 ---
30 root.mainloop()              # イベントループ
```

・スケールのコンストラクタ

Scale(上位ウィジェット, variable=状態保持用オブジェクト)

今回は状態保持用オブジェクトとして DoubleVar オブジェクト vSc1 を与えている。DoubleVar だけでなく Variable クラスのオブジェクトには、値の変化を検知した際に呼び出すコールバック関数を設定（20行目）することができる。

C.2.5.1 Variable クラスのコールバック関数設定

次のように trace メソッドを用いて Variable オブジェクトにコールバック関数を設定することができる。

trace(モード, 関数名)

trace メソッドは、Variable オブジェクトの値に変更などが発生した際に起動するコールバック関数を設定する。モードに 'w' を指定すると値の変更（今回の場合はスケールの値の変更）が発生した際にコールバック関数を起動する。

・プログレスバーのコンストラクタ

Progressbar(上位ウィジェット)

プログレスバー（tkinter.ttk.Progressbar クラスのオブジェクト）には configure メソッドを使用（16行目）することで値を設定することができる。この際、キーワード引数 'value=' に値を与える。

● スケール、プログレスバーの向きと長さ

スケール、プログレスバーのコンストラクタにキーワード引数 'orient=' を与えることで向き（'vertical':上下方向, 'horizontal':左右方向）を設定することができる。また長さは、コンストラクタにキーワード引数 'length=長さ' を与えることで設定する。

● ウィジェットの値の範囲

スケールの値の範囲（下限と上限）を設定するには、コンストラクタにキーワード引数 'from_=下限値', 'to=上限

値'を与える。(from_のアンダースコアに注意) スライダの向きが上下方向の場合は上端が下限値であり、左右方向の場合は左端が下限値となる。

プログレスバーには上限のみ設定することができる。コンストラクタにキーワード引数 'maximum=上限値' を与えることで上限を設定することができる。

C.3 メニューの構築

ここでは図 62 のようなアプリケーションを例に挙げてメニューの構成方法について説明する。

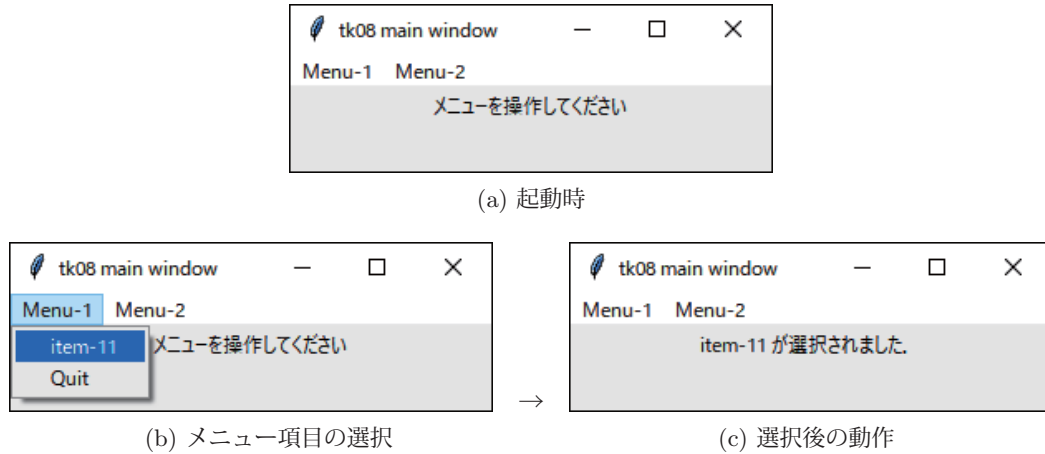


図 62: メニュー

このアプリケーションは、ウィンドウ上部に「Menu-1」「Menu-2」という 2 つのメニューを備えたメニューバーを持つ。それぞれのメニューはプルダウンの形でメニュー項目を表示する。メニュー項目を選択すると、その旨を告げるメッセージをラベルに表示する。「Menu-1」の項目「Quit」を選択することでアプリケーションが終了する。

このアプリケーションの実装例をサンプルプログラム tk08.py に示す。

プログラム：tk08.py

```
1  # coding: utf-8
2  import tkinter          # Tkinter
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
6  root.title( 'tk08 main window' )    # ウィンドウのタイトル
7  root.geometry( '290x50+20+10' ) # ウィンドウのサイズと位置
8
9  frMain = tkinter.Frame(root)    # GUI構築用のコンテナ
10 frMain.pack()    # frMainをrootに配置
11
12 #---
13 # メニューの構築
14 mb1 = tkinter.Menu(root)        # メニューバーの生成
15 root.config( menu=mb1 )        # 最上位ウィジェットへの取り付け
16
17 # メニュー選択を受けて起動する関数
18 def cmdMn11():
19     lb1['text'] = 'item-11 が選択されました。'
20 def cmdMn12():
21     print('Quit が選択されました。終了します。')
22     root.quit()
23
24 def cmdMn21():
25     lb1['text'] = 'item-21 が選択されました。'
26 def cmdMn22():
27     lb1['text'] = 'item-22 が選択されました。'
28
29 # メニュー 1
30 mn1 = tkinter.Menu( mb1, tearoff=0 )
31 mn1.add_command( label='item-11', command=cmdMn11 )    # メニュー項目の登録
32 mn1.add_command( label='Quit', command=cmdMn12 )      # メニュー項目の登録
33 mb1.add_cascade( label='Menu-1', menu=mn1 )          # メニュー 1 をメニューバーに登録
```

```

34
35 # メニュー 2
36 mn2 = tkinter.Menu( mb1, tearoff=0 )
37 mn2.add_command( label='item-21', command=cmdMn21 )      # メニュー項目の登録
38 mn2.add_command( label='item-22', command=cmdMn22 )      # メニュー項目の登録
39 mb1.add_cascade( label='Menu-2', menu=mn2 )              # メニュー 2 をメニューバーに登録
40
41 # ラベル
42 lb1 = tkinter.Label( frMain, text='メニューを操作してください' )
43 lb1.pack( side='top' )
44
45 #--- アプリケーションの起動 ---
46 root.mainloop()          # イベントループ

```

解説：

14 行目で Menu オブジェクト mb1 を生成し、15 行目でそれをメニューバーとして最上位ウィジェット root に取り付けている。この処理においては root に対して config メソッドを使用する。その際、config メソッドのキーワード引数 'menu=' にメニューバーとなるオブジェクトを与える。

メニューバーにメニューを取り付けるには、各メニューとなる Menu オブジェクトを生成して、メニューバーの配下に登録する。各メニューにメニュー項目を登録するには当該メニューオブジェクトに対して add_command メソッドを使用する。このメソッドのキーワード引数 'label=' にはメニュー項目の表示名を与え、'command=' には、そのメニュー項目を選択した際に起動する関数の名前を与える。

C.4 Canvas の描画

図形の描画には Canvas オブジェクトを使用する。

• Canvas のコンストラクタ

Canvas(上位ウィジェット, width=横幅, height=高さ, bg=背景色)

背景色には 3 原色 (RGB) の配合を 16 進数で、'#' で始まる文字列²⁶⁹ として与える。Canvas オブジェクトに対して、以下に示すような各種の描画メソッドを使用して描画する。

C.4.1 描画メソッド (一部)

• 矩形の描画

create_rectangle(x1,y1,x2,y2, width=線幅, outline=線の色, fill=塗りの色)

Canvas オブジェクト上に矩形 (長方形) を描画する。矩形の左上の座標を (x1,y1)、右下の座標を (x2,y2) として描く。

• 楕円の描画

create_oval(x1,y1,x2,y2, width=線幅, outline=線の色, fill=塗りの色)

Canvas オブジェクト上に楕円を描画する。左上の座標を (x1,y1)、右下の座標を (x2,y2) とする矩形に内接する楕円を描く。

• 楕円弧の描画

create_arc(x1,y1,x2,y2, style=スタイル,
width=線幅, outline=線の色, fill=塗りの色, start=開始角, extent=弧の開き角)

Canvas オブジェクト上に楕円弧を描画する。左上の座標を (x1,y1)、右下の座標を (x2,y2) とする矩形に内接する楕円の一部を描く。スタイルには円弧の形状 (図 63 参照) を指定する。角度の単位は度数法²⁷⁰ の「度」で、回転方向は反時計回りである。

²⁶⁹HTML5 の CSS における色指定と同様。

²⁷⁰1 周を 360 度とする角度の表現。



図 63: 楕円弧のスタイル

• 多角形の描画

`create_polygon(x1,y1,x2,y2,...,xn,yn, width=線幅, outline=線の色, fill=塗りの色)`

Canvas オブジェクト上に多角形を描画する。頂点の座標を $(x1,y1),(x2,y2),\dots,(xn,yn)$ で与える。

• 折れ線の描画

`create_line(x1,y1,x2,y2,...,xn,yn, width=線幅, fill=線の色)`

Canvas オブジェクト上に折れ線を描画する。始点 $(x1,y1)$ ではじまり終点 (xn,yn) で終わる頂点の座標の列 $(x1,y1),(x2,y2),\dots,(xn,yn)$ で与える。

• 文字列の描画

`create_text(x,y, text=文字列, font=フォント指定, anchor=位置の基準, fill=色)`

Canvas オブジェクト上に文字列を描画する。描画位置を (x,y) で指定する。この際、表示する文字列全体のどの位置を基準にするかを位置の基準（図 64）で指定する。

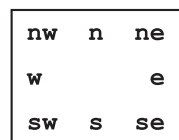


図 64: 位置の基準（anchor）

フォント指定に文字列を表示するためのフォント名とサイズなどを指定する。

C.4.1.1 使用できるフォントを調べる方法

使用できるフォント名を調べるためのプログラム例 `tk_font01.py` を示す。

プログラム：tk_font01.py

```
1 # coding: utf-8
2 import tkinter          # Tkinter
3 import tkinter.font as tkfont  # フォント関連モジュール
4
5 root = tkinter.Tk()      # アプリケーションのウィジェット
6
7 fntL = list( tkfont.families(root) )  # フォントリストの取得
8 for f in fntL:           # フォント名の出力
9     print( f )
```

この例のように、`tkinter.font` モジュールの `families` 関数を使用するとフォント名の列が得られる。

tk_font01.py の実行例

```
C:\Users\katsu > py tk_font01.py
System
@System
Terminal
...
(途中省略)
...
Highlight LET
Odessa LET
Scruff LET
```

・画像の描画

`create_image(x,y, image=画像オブジェクト, anchor=位置の基準)`

Canvas オブジェクト上に画像オブジェクトを描画する。描画位置を (x,y) で指定する。この際、表示する画像オブジェクトのどの位置を基準にするかを**位置の基準** (図 64) で指定する。

Tkinter で使用できる画像フォーマットの種類は限られており、多くの画像フォーマットを取り扱うには Pillow ライブラリ (画像処理用ライブラリ) を併用するのが良い。具体的には、Pillow の `open` 関数で画像を読み込み、それを Pillow の `PhotoImage` 関数によって Tk で使用できる画像オブジェクトに変換²⁷¹ する。(後のサンプルプログラム `tk09.py` で使用例を示す)

Canvas 上に各種の描画を行うサンプルプログラム `tk09.py` を次に示す。

プログラム：tk09.py

```
1  # coding: utf-8
2  import tkinter                                # Tkinter
3  import PIL.Image as PilImage                  # Pillowのimageモジュール
4  import PIL.ImageTk as PilImageTk             # PillowのImageTkモジュール
5
6  #--- GUIの構築 ---
7  root = tkinter.Tk()                          # アプリケーションの最上位ウィジェット
8  root.title( 'tk09 main window' )            # ウィンドウのタイトル
9  root.geometry( '400x300+20+10' )            # ウィンドウのサイズと位置
10
11 frMain = tkinter.Frame(root)                 # GUI構築用のコンテナ
12 frMain.pack()                               # frMainをrootに配置
13
14 #--- Canvas描画 ---
15
16 # Canvasの生成と配置
17 cv1 = tkinter.Canvas(frMain,bg='ffffff',width=400,height=300)
18 cv1.pack( side='top' )
19
20 # 楕円
21 cv1.create_oval(10,10,150,100,
22                outline='ff0000',width=10,fill='00ff00')
23 # 矩形
24 cv1.create_rectangle(170,10,270,100,
25                      outline='0000ff',width=10,fill='ffff00')
26 # 多角形
27 cv1.create_polygon(290,10,380,10,380,100,335,40,290,100,
28                    outline='000000',width=10,fill='ff00ff')
29 # 折れ線
30 cv1.create_line(10,120,380,120,10,150,380,150,
31                 width=10,fill='00ffff')
32 # 文字列
33 cv1.create_text(10,170,text='ゴシック',
34                 font='IPAゴシック 40', anchor='nw', fill='000000')
35 cv1.create_text(10,230,text='明朝',
36                 font='IPA明朝 40', anchor='nw', fill='000000')
37 # 楕円弧
38 cv1.create_arc(120,230,285,330, style='pieslice',
39                start=60, extent=120, outline='000000',width=5,fill='ffff00')
40
41 # 画像 (Pillowオブジェクトとして読み込み, Tkオブジェクトに変換)
42 im0 = PilImage.open('Earth2.jpg')            # Pillowオブジェクト
43 im1 = PilImageTk.PhotoImage( im0.resize((130,130)) ) # Tkオブジェクト
44 cv1.create_image(250,160,image=im1,anchor='nw')
45
46 #--- アプリケーションの起動 ---
47 root.mainloop()                             # イベントループ
```

このプログラムを実行した例を図 65 に示す。

²⁷¹ 注意) Pillow の `PhotoImage` 関数で生成した画像オブジェクトには**ガベージコレクション**が予期せぬタイミングで働くことがある。特に当該オブジェクトを関数内部のローカル変数 (局所変数) に格納した場合において画像オブジェクトが予期せず消滅してしまうことがある。このため、当該関数で生成したオブジェクトは、グローバル変数に格納するなどして、予期せぬ消滅を防ぐ必要がある。



図 65: Canvas 上での各種描画

C.4.2 図形の管理

Canvas 上に描く図形には、描画メソッドの実行の際にキーワード引数 `'tag=タグ名'` を指定することでタグ（識別名）を与えることができ、これを指定して Canvas 上での位置の変更や削除などの操作ができる。例えば、Canvas オブジェクト `cv1` 上の図形 `'tg01'` を削除するには

```
cv1.delete('tg01')
```

のように `delete` メソッドを実行する。また、描画位置を変更するには `coords` メソッドを用いる。

描画した図形を削除する例をサンプルプログラム `tk09-2.py` に、位置を移動する例をサンプルプログラム `tk09-3.py` にそれぞれ示す。

プログラム：tk09-2.py

```
1  # coding: utf-8
2  # モジュールの読み込み
3  import tkinter                # Tkinter
4
5  #--- GUIの構築 ---
6  root = tkinter.Tk()           # アプリケーションの最上位ウィジェット
7  root.title( 'tk09-2 main window' ) # ウィンドウのタイトル
8  root.geometry( '400x200+20+10' ) # ウィンドウのサイズと位置
9  root.resizable(False,False)
10
11 # コールバック：図形追加
12 def cmdGen():
13     cv1.create_oval(10,10,390,130, tag='tg01',
14                     outline='#ff0000',width=10,fill='#00ff00')
15 # コールバック：図形削除
16 def cmdDel():
17     cv1.delete('tg01')
18
19 btnGen = tkinter.Button(root,text='追加',command=cmdGen)    # 追加ボタン
20 btnGen.pack( fill='x' )
21
22 btnDel = tkinter.Button(root,text='削除',command=cmdDel)    # 削除ボタン
23 btnDel.pack( fill='x' )
24
25 # Canvasの生成と配置
26 cv1 = tkinter.Canvas(root,bg='#ffffff')
27 cv1.pack( fill='both' )
28
29 #--- アプリケーションの起動 ---
30 root.mainloop()      # イベントループ
```

このプログラムでは「追加」ボタンのクリックにより描画を、「削除」ボタンのクリックにより図形の削除を実行する。(図 66)



図 66: Canvas 上の図形の追加と削除

プログラム：tk09-3.py

```

1  # coding: utf-8
2  import tkinter          # Tkinter
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
6  root.title( 'tk09-3 main window' ) # ウィンドウのタイトル
7  root.geometry( '350x250+20+10' )   # ウィンドウのサイズと位置
8  root.resizable(False,False)
9
10 # コールバック：図形追加
11 def cmdUp():              # ↑
12     global x1, y1, x2, y2
13     y1 -= 5;      y2 -= 5
14     cv1.coords('tg01',x1,y1,x2,y2)
15 def cmdLeft():            # ←
16     global x1, y1, x2, y2
17     x1 -= 5;      x2 -= 5
18     cv1.coords('tg01',x1,y1,x2,y2)
19 def cmdRight():           # →
20     global x1, y1, x2, y2
21     x1 += 5;      x2 += 5
22     cv1.coords('tg01',x1,y1,x2,y2)
23 def cmdDown():            # ↓
24     global x1, y1, x2, y2
25     y1 += 5;      y2 += 5
26     cv1.coords('tg01',x1,y1,x2,y2)
27
28 btnUp = tkinter.Button(root,text='↑',command=cmdUp)      # ↑ ボタン
29 btnUp.grid( row=0, column=1 )
30
31 btnLeft = tkinter.Button(root,text='←',command=cmdLeft)  # ← ボタン
32 btnLeft.grid( row=1, column=0 )
33
34 btnRight = tkinter.Button(root,text='→',command=cmdRight) # → ボタン
35 btnRight.grid( row=1, column=2 )
36
37 btnDown = tkinter.Button(root,text='↓',command=cmdDown)  # ↓ ボタン
38 btnDown.grid( row=2, column=1 )
39
40 # Canvasの生成と配置
41 cv1 = tkinter.Canvas(root,bg='ffffff', width=300,height=190)
42 cv1.grid( row=1, column=1 )
43
44 x1 = 125;    y1 = 70
45 x2 = 175;    y2 = 120
46 cv1.create_oval(x1,y1,x2,y2, tag='tg01',
47                 outline='ff0000',width=9,fill='#00ff00')
48
49 #--- アプリケーションの起動 ---
50 root.mainloop()      # イベントループ

```

このプログラムでは「↑」「←」「→」「↓」の各ボタンのクリックにより図形の位置を変更する。(図 67)



図 67: Canvas 上での図形の移動

C.5 イベントハンドリング

各種のウィジェットには、受信すべきイベントを登録し、それに対して起動する関数（コールバック関数もしくはイベントハンドラ）を設定することができる。これには次に示す `bind` メソッドを使用する。

・ウィジェットへのイベントハンドリングの実装

`bind(イベントシーケンス, コールバック関数の名前)`

イベントシーケンスで示すイベントが発生した際にコールバック関数を起動する。イベントシーケンスは '`< ... >`' でイベント名を括った記述の連鎖²⁷²である。

キーボードとマウスに関するイベントを表 54 に示す。

表 54: イベントの表記（一部）

キーボード		マウス	
表記	意味	表記	意味
<code>KeyPress / Key</code>	キーの押下	<code>ButtonPress / Button</code>	ボタンの押下
<code>KeyRelease</code>	キーの解放	<code>ButtonRelease</code>	ボタンの解放
		<code>Motion</code>	マウスの移動

これらイベントには各種のモディファイア（装飾）をつけることができる。（表 55）

表 55: モディファイア（一部）

表記	意味	表記	意味
<code>Control</code>	CTRL キー	<code>Shift</code>	SHIFT キー
<code>Alt</code>	ALT キー		
<code>Button1</code>	マウスの左ボタン	<code>Button3</code>	マウスの右ボタン
<code>Double</code>	ダブルクリック	<code>Triple</code>	トリプルクリック

テキストウィジェットとラベルウィジェットに対してイベントハンドリングを実装するプログラムの例 `tk10.py` を示す。

プログラム：tk10.py

```

1  # coding: utf-8
2  import tkinter          # Tkinter
3
4  #--- GUIの構築 ---
5  root = tkinter.Tk()      # アプリケーションの最上位ウィジェット
6  root.title( 'tk10 main window' ) # ウィンドウのタイトル
7  root.geometry( '600x140+20+10' ) # ウィンドウのサイズと位置
8
9  frMain = tkinter.Frame(root) # GUI構築用のコンテナ
10 frMain.pack() # frMainをrootに配置
11
12 # テキスト
13 txt = tkinter.Text(frMain, width=40, height=10, wrap='none')
14 txt.pack( side='left' )
15
16 # ラベル

```

²⁷²複雑なイベントの連鎖をハンドリングできる。詳しくは Tkinter の公式ドキュメントを参照のこと。

```

17 lb1 = tkinter.Label(frMain,width=300,height=150,bg='#cccccc')
18 lb1.pack( side='left' )
19
20 #--- イベントハンドリング ---
21
22 # コールバック関数
23 def evAltKey( ev ):
24     msg = '● ALTキー:\t\t' + str(ev.time) + ',\t' + ev.keysym +\
25           ',\t' + ev.type
26     print(msg)
27 def evKey( ev ):
28     msg = '● キー入力:\t\t' + str(ev.time) + ',\t' + ev.keysym +\
29           ',\t' + ev.type
30     print(msg)
31
32 def evSClick1( ev ):
33     msg = '● 左シングルクリック:\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
34           ',\t' + ev.type
35     print(msg)
36 def evSClick3( ev ):
37     msg = '● 右シングルクリック:\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
38           ',\t' + ev.type
39     print(msg)
40 def evDbClick1( ev ):
41     msg = '● 左ダブルクリック:\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
42           ',\t' + ev.type
43     print(msg)
44 def evDbClick3( ev ):
45     msg = '● 右ダブルクリック:\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
46           ',\t' + ev.type
47     print(msg)
48
49 def evDrag1( ev ):
50     msg = '● 左ドラッグ:\t\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
51           ',\t' + ev.type
52     print(msg)
53 def evDrag3( ev ):
54     msg = '● 右ドラッグ:\t\t' + str(ev.time) + ',\t' + str((ev.x,ev.y)) +\
55           ',\t' + ev.type
56     print(msg)
57
58 # イベント処理の登録
59 txt.bind('<Alt-Key>',evAltKey)
60 txt.bind('<Key>',evKey)
61
62 lb1.bind('<Button-1>',evSClick1)
63 lb1.bind('<Button-3>',evSClick3)
64 lb1.bind('<Double-Button-1>',evDbClick1)
65 lb1.bind('<Double-Button-3>',evDbClick3)
66
67 lb1.bind('<Button1-Motion>',evDrag1)      # 左ボタンでドラッグ
68 lb1.bind('<Button3-Motion>',evDrag3)     # 右ボタンでドラッグ (Windows)
69
70 #--- アプリケーションの起動 ---
71 root.mainloop()                        # イベントループ

```

このプログラムを実行すると、図 68 のようなウィンドウが表示される。59,60 行目でテキストウィジェットオブジェクト txt に対してキーボードイベントのハンドリングを設定している。62~68 行目ではラベルウィジェット lb1 に対してマウスイベントのハンドリングを設定している。

キーボード入力やマウス操作を行うと、それを標準出力に報告する。実行例を次に示す。

実行例

```

● キー入力:      141868953,  a,      2
● キー入力:      141879187,  Alt_L,  2
● ALT キー:      141879296,  x,      2
● 左シングルクリック: 141883453,  (19, 22),  4
● 左ドラッグ:    141883656,  (20, 23),  6
● 左ドラッグ:    141883671,  (20, 24),  6
● 左ドラッグ:    141883687,  (21, 24),  6

```



図 68: キーボードとマウスに反応するアプリケーション

コールバック関数は引数としてイベントオブジェクトを受け取る。このオブジェクトのプロパティにイベントに関する各種の情報が保持されている。(表 56)

表 56: イベントオブジェクトのプロパティ (一部)

表記	意味	表記	意味
widget	当該ウィジェット	time	時刻
type	イベントのタイプ		
x	ウィジェット上でのマウスの x 座標	y	ウィジェット上でのマウスの y 座標
x_root	ウィンドウ上でのマウスの x 座標	y_root	ウィンドウ上でのマウスの y 座標
keycode	キーコード	keysym	キーシンボル

C.5.1 時間を指定した関数の実行

指定した時間が経過した後²⁷³ でコールバック関数を呼び出すには、after メソッドを使用する。このメソッドはウィジェットオブジェクトに対して実行する。

・タイマー

after(経過時間, コールバック関数の名前)

経過時間はミリ秒の単位の数値で与える。コールバック関数は引数を取らない。

after メソッドを応用して時計を実現したプログラムの例 tk11.py を示す。

プログラム: tk11.py

```

1  # coding: utf-8
2  import tkinter                # Tkinter
3  from datetime import datetime # datetime
4
5  #--- GUIの構築 ---
6  root = tkinter.Tk()           # アプリケーションの最上位ウィジェット
7  root.title( 'tk11 main window' ) # ウィンドウのタイトル
8  root.geometry( '280x60+20+10' ) # ウィンドウのサイズと位置
9
10 frMain = tkinter.Frame(root)   # GUI構築用のコンテナ
11 frMain.pack()                  # frMainをrootに配置
12
13 # ラベル
14 lb1 = tkinter.Label(frMain,width=300,height=150,
15                      bg='#dddddd',font='IPAゴシック 18')
16 lb1.pack( side='top' )
17
18 #--- イベントハンドリング ---
19
20 # コールバック関数
21 def evFunc1( ):
22     d = datetime.now()
23     msg = str(d.year) + '年' + str(d.month) + '月' + str(d.day) + '日\n' + \
24           str(d.hour) + '時' + str(d.minute) + '分' + str(d.second) + '秒'
25     lb1['text'] = msg
26     lb1.after(1000, evFunc1)
27

```

²⁷³ タイマーイベントと呼ばれることが多い

```

28 lb1.after(500, evFunc1)
29
30 #--- アプリケーションの起動 ---
31 root.mainloop()          # イベントループ

```

このプログラムを実行すると、図 69 のようなウィンドウが表示されて時刻を知らせる。

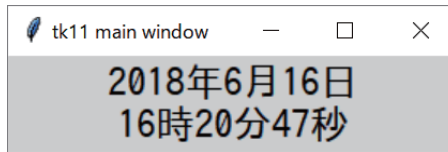


図 69: 時計のアプリケーション

C.6 複数のウィンドウの表示

Tk() の実行によりアプリケーションの最上位のウィジェットを生成し、それを当該アプリケーションのメインウィンドウとするが、これとは別のウィンドウを生成するには Toplevel() を実行する。2 つ目のウィンドウを生成して表示するプログラムの例を tk12.py に示す。

プログラム：tk12.py

```

1  # coding: utf-8
2  import tkinter          # Tkinter
3
4  #--- メインウィンドウ ---
5  root = tkinter.Tk()
6  root.title( 'tk12 main window' )    # メインウィンドウのタイトル
7  root.geometry( '280x60+20+10' )    # メインウィンドウのサイズと位置
8
9  lb1 = tkinter.Label(root, text='メインウィンドウ')
10 lb1.pack( fill='both', expand=1 )
11
12 #--- サブウィンドウ ---
13 subW = tkinter.Toplevel()
14 subW.title('tk12 sub window')      # サブウィンドウのタイトル
15 subW.geometry( '280x50+120+110' )  # サブウィンドウのサイズと位置
16
17 lb2 = tkinter.Label(subW, text='サブウィンドウ')
18 lb2.pack( fill='both', expand=True )
19
20 #--- アプリケーションの起動 ---
21 root.mainloop()          # イベントループ

```

このプログラムを実行した例を図 70 に示す。

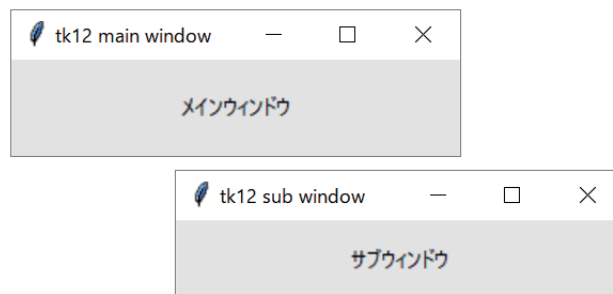


図 70: 複数のウィンドウ

このプログラムはメインウィンドウを閉じると、サブウィンドウ（プログラム中の subW）を含めて全て終了する。

C.7 ディスプレイやウィンドウに関する情報の取得

Tk() の実行によって生成したウィンドウ（最上位ウィジェット）からは様々な情報が取得できる。最上位ウィジェット root がある場合に、それに対して実行するメソッドと得られる値を表 57 に示す。

表 57: ディスプレイやウィンドウに関する情報を取得するメソッド

メソッド	得られる値	メソッド	得られる値
wininfo_screenwidth()	ディスプレイの横幅	wininfo_screenheight()	ディスプレイの高さ
wininfo_width()	ウィンドウの横幅	wininfo_height()	ウィンドウの高さ
wininfo_x()	ウィンドウの横位置	wininfo_y()	ウィンドウの縦位置

スクリーンのサイズ、ウィンドウのサイズと位置を取得するサンプルプログラム tk13.py を示す。

プログラム：tk13.py

```
1  # coding: utf-8
2  import tkinter          # Tkinterの読み込み
3
4  #--- コールバック関数 ---
5  def cmdConfig(ev):
6      w = root.wininfo_width()      # 横幅の取得
7      h = root.wininfo_height()     # 高さの取得
8      x = root.wininfo_x()          # ウィンドウの横位置
9      y = root.wininfo_y()          # ウィンドウの縦位置
10     msg = str(w) + 'x' + str(h) + '+' + str(x) + '+' + str(y)
11     lb1['text'] = msg
12
13  #--- GUIの構築 ---
14  root = tkinter.Tk()            # アプリケーションの最上位ウィジェット
15  root.title('tk13 main window') # ウィンドウのタイトル
16  root.geometry('400x100+20+10')  # ウィンドウのサイズと位置
17  root.bind('<Configure>', cmdConfig)
18
19  # 配置テスト用のラベル
20  lb1 = tkinter.Label( root, text='ウィンドウサイズ',
21                      font='IPAゴシック 18',
22                      bg='#000000', fg='#ffffff' )
23  lb1.pack( fill='both', expand=True )
24
25  W = root.wininfo_screenwidth()   # スクリーンの幅
26  H = root.wininfo_screenheight()  # スクリーンの高さ
27  print('画面サイズ：',W,'x',H)
28
29  #--- アプリケーションの起動 ---
30  root.mainloop()                # イベントループ
```

ウィンドウサイズの変更などはイベント 'Configure' としてハンドリング（17行目）している。

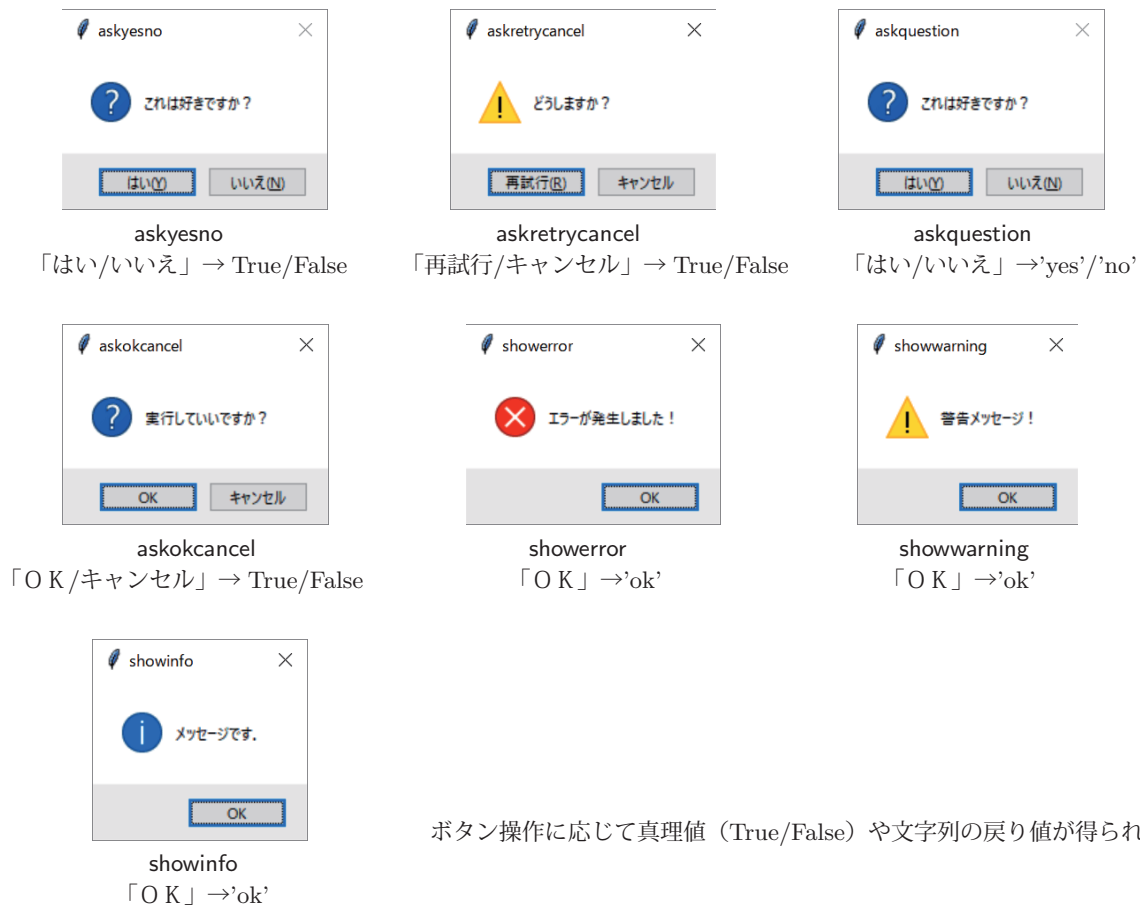
プログラムを実行して、表示されたウィンドウの位置とサイズを変更する様子を図 71 に示す。



図 71: ウィンドウの移動とサイズ変更に伴う情報の取得

C.8 メッセージボックス (messagebox)

「ダイアログボックス」と呼ばれることの多い UI が Tkinter でもメッセージボックス (messagebox) という形で提供されている。Tkinter のメッセージボックスは tkinter.messagebox モジュール下の関数として提供されている。各種メッセージボックスを表示するための関数の名称と表示例、ボタン操作の結果得られる関数の戻り値を図 72 に示す。



ボタン操作に応じて真理値 (True/False) や文字列の戻り値が得られる。

図 72: 各種メッセージボックス： 表示するための関数の名称とボタン操作に対する戻り値

図 72 のようなメッセージボックスを表示するサンプルプログラムを tk14.py に示す。

プログラム：tk14.py

```
1 # coding: utf-8
2 import tkinter
3 from tkinter import messagebox
4
5 #--- コールバック関数 ---
6 def cb_yesno():
7     r = messagebox.askyesno(
8         title='askyesno', message='これは好きですか？' )
9     vEnt.set( str(r)+' : '+str(type(r)) )
10 def cb_retrycancel():
11     r = messagebox.askretrycancel(
12         title='askretrycancel', message='どうしますか？' )
13     vEnt.set( str(r)+' : '+str(type(r)) )
14 def cb_question():
15     r = messagebox.askquestion(
16         title='askquestion', message='これは好きですか？' )
17     vEnt.set( str(r)+' : '+str(type(r)) )
18 def cb_okcancel():
19     r = messagebox.askokcancel(
20         title='askokcancel', message='実行していいですか？' )
21     vEnt.set( str(r)+' : '+str(type(r)) )
22 def cb_showerror():
23     r = messagebox.showerror(
24         title='showerror', message='エラーが発生しました！' )
25     vEnt.set( str(r)+' : '+str(type(r)) )
```

```

26 def cb_showwarning():
27     r = messagebox.showwarning(
28         title='showwarning', message='警告メッセージ！' )
29     vEnt.set( str(r)+' : '+str(type(r)) )
30 def cb_showinfo():
31     r = messagebox.showinfo(
32         title='showinfo', message='メッセージです.' )
33     vEnt.set( str(r)+' : '+str(type(r)) )
34 def cb_close():
35     r = messagebox.askokcancel(
36         title='askokcancel', message='終了しようとしています！' )
37     print('戻り値:',r,':',type(r))
38     if r:
39         root.destroy()
40
41 #--- ウィンドウの構築 ---
42 root = tkinter.Tk()
43 root.title('tk14')
44 root.geometry('350x300+20+10')
45
46 # 各種メッセージボックスを表示するボタン
47 b1 = tkinter.Button(root,text='askyesnoのテスト',command=cb_yesno)
48 b1.pack(side='top',fill='both',expand=True)
49
50 b2 = tkinter.Button(root,text='askretrycancelのテスト',command=cb_retrycancel)
51 b2.pack(side='top',fill='both',expand=True)
52
53 b3 = tkinter.Button(root,text='askquestionのテスト',command=cb_question)
54 b3.pack(side='top',fill='both',expand=True)
55
56 b4 = tkinter.Button(root,text='askokcancelのテスト',command=cb_okcancel)
57 b4.pack(side='top',fill='both',expand=True)
58
59 b5 = tkinter.Button(root,text='showerrorのテスト',command=cb_showerror)
60 b5.pack(side='top',fill='both',expand=True)
61
62 b6 = tkinter.Button(root,text='showwarningのテスト',command=cb_showwarning)
63 b6.pack(side='top',fill='both',expand=True)
64
65 b7 = tkinter.Button(root,text='showinfoのテスト',command=cb_showinfo)
66 b7.pack(side='top',fill='both',expand=True)
67
68 # エントリ
69 vEnt = tkinter.StringVar()
70 vEnt.set('ここに戻り値が表示されます')
71 e1 = tkinter.Entry(root,textvariable=vEnt)
72 e1.pack(side='top',fill='both',expand=True)
73
74 # ラベル
75 lb = tkinter.Label(root,text='ウィンドウを閉じてみてください.',
76                     font='18')
77 lb.pack(side='top',fill='both',expand=True)
78
79 #--- ウィンドウを閉じる際のコールバック設定 ---
80 root.protocol('WM_DELETE_WINDOW', cb_close)
81
82 #--- イベントループ ---
83 root.mainloop()

```

このプログラムを実行すると図 73 のようなウィンドウが表示される。

ウィンドウ内のボタン操作に対するコールバック処理として、各種のメッセージボックスを表示するための関数を呼び出している。メッセージボックスのボタン操作に対する関数の戻り値とそのデータ型をウィンドウ内の Entry（テキストボックス）に表示する。

C.8.1 アプリケーション終了のハンドリング

通常の場合、GUI アプリケーションはウィンドウを閉じると終了するが、ウィンドウを閉じる操作に対するコールバック処理を設定することができる。先のサンプルプログラム tk14.py の 80 行目で、ウィンドウ root に対して

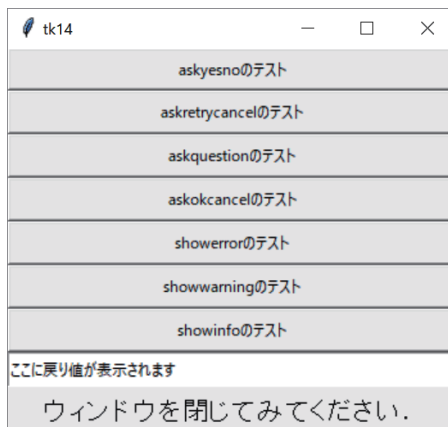


図 73: アプリケーションのメインウィンドウ

protocol メソッドを実行しているが、その際にハンドリングのためのコールバック関数 `cb_close` (34～39 行目で定義) を設定している。これにより、ウィンドウを閉じる操作をするとメッセージボックスを表示して、本当に終了するか否かの選択をユーザに促す。

`tk14.py` の 39 行目にあるように、ウィンドウに対して `destroy` メソッドを実行すると当該ウィンドウが閉じられる。

C.9 ウィジェットの親、子を調べる方法

あるウィジェットが属する上位のウィジェット（親ウィジェット）は、当該ウィジェットの `master` 属性が保持している。また、あるウィジェットが配下に持つウィジェットは、当該ウィジェットに `wininfo_children` メソッドを実行することでリストの形で得られる。

`tkinter.Tk()` で作成した最上位のウィジェットは親ウィジェットを持たず、`master` 属性を参照すると `None`（ヌルオブジェクト）となる。また、最下位のウィジェット（配下に子を持たないウィジェット）に対して `wininfo_children` メソッドを実行すると空リスト `[]` が得られる。

【サンプルプログラム】

次に示すサンプルプログラムで定義されている関数 `dispWtree` は、第一引数にウィジェットのリストを、第二引数に `0` を与えて実行するとウィジェットの階層関係を表示する。

プログラム（一部）：これを `tk08.py` の末尾²⁷⁴ に追加して実行する

```
1 def dispWtree(wl,n):      # ウィジェットリスト wl の階層を表示する関数
2     fl = ' '*n*4
3     for w in wl:
4         print( fl, type(w), ': ', w )
5         wl2 = w.wininfo_children()
6         dispWtree(wl2,n+1)
7
8 dispWtree([root],0)      # 最上位のウィジェット root 以下の階層を調べる
```

これを先に示したプログラム `tk08.py` (p.419) の末尾に追加して実行すると、メニュー操作「Quit」で終了する際に、次のようにウィジェット階層を表示する。

実行結果の例.

```
<class 'tkinter.Tk'> : .
  <class 'tkinter.Frame'> : .!frame
    <class 'tkinter.Label'> : .!frame.!label
  <class 'tkinter.Menu'> : .!menu
    <class 'tkinter.Menu'> : .!menu.!menu
    <class 'tkinter.Menu'> : .!menu.!menu2
```

ウィジェットの種類と共に `root` 以下の階層関係が表示されている。

²⁷⁴実行時にエラーが出る場合は、関数 `dispWtree` の定義の記述の部分を「`root.mainloop()`」の直前に挿入されたい。

D ライブラリの取り扱いについて

D.1 ライブラリの読み込みに関すること

ライブラリを読み込むための最も基本的な記述は

```
import ライブラリ名
```

である。当該ライブラリに含まれるクラスやメソッド、関数などを使用するには、それらの名前の前に

‘ライブラリ名.’

を付ける。(接頭辞の付加) 例えば math モジュールの sin 関数を呼び出す場合は、

```
import math
y = math.sin( math.pi/2 )
```

などとする。

パッケージ構成のライブラリの場合は、1つのパッケージが複数のモジュールから構成されている。例えば「パッケージ1」というパッケージが「モジュール1」というモジュールを保持している場合は、次のようにしてそのモジュールを読み込む。

```
import パッケージ 1. モジュール 1
```

この後、例えば次のようにして、そこに属する関数やクラスを利用することができる。

```
パッケージ 1. モジュール 1. 関数 (引数,...)
```

パッケージは、サブパッケージから成る階層構造をとる場合もある。その場合は、

```
import パッケージ 1. サブパッケージ. サブサブパッケージ... モジュール
```

のようにして読み込む。ただし、この方法でモジュールを読み込むと、その利用において、接頭辞のドット「.」による連結表記が長くなってしまい、記述の煩わしさが生じる。そのような場合は、モジュールの読み込み時に別名を与えると良い。

D.1.1 ライブラリ読み込みにおける別名の付与

先に例示したような、ドット表記の長いパッケージ読み込みにおいては、別名を与えて簡潔な記述にすることができる。具体的には、import による読み込みの最後に「as 別名」を記述する。

```
import パッケージ 1. サブパッケージ. サブサブパッケージ... モジュール as 別名
```

この後、例えば次のようにして簡潔な記述ができる。

```
別名. 関数 (引数,...)
```

「別名」は任意に決めて良い。

D.1.2 接頭辞を省略するためのライブラリの読み込み

import でライブラリを読み込む際に from を使って

```
from ライブラリの指定 import 関数やクラスなど
```

として特定の関数やクラスを指定して読み込むことができる。例えば、math モジュールの sin と pi のみを読み込んで使用するには次のようにする。

```
from math import sin, pi
```

これにより、sin と pi の使用に際して接頭辞 ‘math.’ を付ける必要がなくなる。(次の例)

例. 接頭辞を省略するためのモジュール読み込み

```
>>> from math import sin, pi [Enter] ← math モジュールから sin と pi のみ読み込む
>>> sin( pi/2 ) [Enter] ← 接頭辞無しで計算実行
1.0 ← 結果表示
```

更に、読み込み時の関数名やクラス名の代わりにアスタリスク「*」を指定すると、当該モジュール内のすべてのものが読み込まれ、接頭辞を付ける必要が一切無くなる。(次の例参照)

例. モジュール内のすべてのものを読み込む

```
>>> from math import * Enter ← math モジュールから全てを読み込む
>>> acos( sin( 3 * pi / 2 ) ) Enter ← 接頭辞無しで計算実行
3.141592653589793 ← 結果表示
```

接頭辞を付けずに math モジュールの acos, sin, pi が使用できることがわかる。

D.1.2.1 接頭辞を省略する際の注意（名前の衝突）

ライブラリの読み込みにおいてアスタリスク「*」を使用する場合は**名前の衝突**に注意しなければならない。このことは特に複数のライブラリを併用する場合に重要である。

複数の異なるライブラリが同一の関数名やクラス名を使用している場合に名前の衝突が起こる。例えば math モジュールと mpmath ライブラリは同じ名前の関数を多数提供しており、この問題が発生する。具体的な例を挙げると、sin 関数は両方のライブラリが提供しており、アスタリスクによる読み込みによって問題が発生する。（次の例参照）

例. math モジュールと mpmath の間での名前の衝突

```
>>> from math import * Enter ← math モジュールから全てを読み込む
>>> sin(pi/2) Enter ← 接頭辞無しで計算実行
1.0 ← math.sin による結果
>>> from mpmath import * Enter ← mpmath から全てを読み込む
>>> sin(pi/2) Enter ← 接頭辞無しで計算実行
mpf('1.0') ← mpmath.sin による結果
```

これは、後から読み込まれた mpmath によって、先に読み込まれたモジュールの関数が上書きされてしまった例である。接頭辞を省略する形でのライブラリの利用に際しては十分な注意が必要である。長い接頭辞を付けることが煩わしい場合は、別名を付与して名前の衝突に対処するのが良い。

D.1.3 ライブラリのパス：sys.path

Python 処理系が読み込むライブラリのパスは sys.path にリストの形で登録されている。

例. sys.path の内容を調べる

```
>>> import sys Enter ← sys モジュールの読み込み
>>> sys.path Enter ← sys.path の内容を参照する
['', 'C:\Program Files\Python311\python311.zip', 'C:\Program Files\Python311\DLLs',
... (途中省略) ...
'C:\Program Files\Python311\Lib\site-packages']
```

この例のように、ライブラリを検索するパスがリストの形で得られる。

利用者が独自に作成したライブラリを Python 処理系で import 可能にするには、当該ライブラリが属するディレクトリのパスを sys.path に追加する²⁷⁵。

D.1.4 既に読み込まれているライブラリの調査： sys.modules

Python 処理系に既に読み込まれているライブラリの一覧は sys モジュールの modules から得られる。この値は辞書型であり、要素のキーはモジュール名、それに対する値は module オブジェクト²⁷⁶である。これを応用して、処理系に読み込まれているモジュールのリストを作成する例を示す。

²⁷⁵JupyterNotebook など一部の対話環境では、sys.path に変更を加えた後、セッションの再起動が必要となる場合もあるので注意すること。

²⁷⁶module オブジェクトに関しては、Python の公式ドキュメントを参照のこと。

例. 処理系に読み込まれているライブラリを確認する

```
>>> import sys      Enter      ← sys モジュールの読み込み
>>> for n,x in enumerate(mlst:=list(sys.modules.keys())): Enter      ←モジュール名の出力ループ
...     print(f' {x+","}<36s}',end=' '); Enter
...     if n%2: print() Enter
... Enter      ←ループの記述の終了
sys,                                builtins,
_frozen_importlib, ... (途中省略) ... _imp,
tokenize,                           linecache,
inspect,                             rlcompleter,
```

ライブラリは他のライブラリを用いて (import して) 作成されていることも多く, そのようなライブラリを読み込むと, 副次的に読み込んだライブラリの名前も `sys.modules` に反映される. あるライブラリが, そのスコープ内で読み込んだライブラリはグローバルのスコープにはなく, 副次的に読み込まれたライブラリの機能は, 通常の形では Python の対話モードでは使用できない.

`sys.modules` から得られるライブラリ名の内, グローバルのスコープで利用できるものを探すには `globals()`²⁷⁷ の戻り値との共通部分を算出する. (次の例参照)

例. グローバルのスコープで読み込まれているモジュールの調査 (先の例の続き)

```
>>> gmlst = set(mlst).intersection( set(globals()) ) Enter      ←モジュール名のセットを取得
>>> print( gmlst ) Enter      ←モジュール名のセットを表示
{'sys'}      ←結果
```

この例の結果は, `sys` モジュールがグローバルのスコープで利用できることを示している.

D.1.5 同一ライブラリの複数回の読み込みに関すること

Python 言語処理系は, `import` で読み込まれたライブラリを `sys.modules` で管理している. これにより, 一旦読み込まれたライブラリは, 後で再度 `import` で読み込み処理をしても再度読み込まれることはない. このことを例を挙げて解説する.

次に示すプログラム `MyLibrary.py` をモジュールとして読み込むと, モジュール内の変数 `v_local` に 123 という整数値が初期値として設定される. モジュールの関数 `getV`, `setV` はその変数の値の参照と設定をするものである.

プログラム: `MyLibrary.py`

```
1 v_local = 123
2
3 def setV(v):
4     global v_local
5     v_local = v
6
7 def getV():
8     global v_local
9     return v_local
```

このモジュールの動作を確認する. (次の例)

例. モジュールの機能を使う

```
>>> import MyLibrary Enter      ←モジュールの読み込み
>>> MyLibrary.getV() Enter      ←モジュール内変数の参照
123      ←値が参照できている
>>> MyLibrary.setV(456) Enter      ←モジュール内変数に新たな値を設定
>>> MyLibrary.getV() Enter      ←値の確認
456      ←値が変更されている
```

この状態で再度同じモジュールを読み込む.

²⁷⁷ 「4.20 使用されているシンボルの調査」(p.339) で解説している.

例. 同一モジュールを再度読み込む（先の例の続き）

```
>>> import MyLibrary as MyLib1  Enter    ←同じモジュールを再度読み込む
>>> MyLib1.getV()  Enter    ←モジュール内変数の参照
456                ← 2 回目のモジュール読み込み以前の値が参照できている
>>> MyLib1.setV(789)  Enter    ←モジュール内変数に新たな値を設定
>>> MyLibrary.getV()  Enter    ←初回読み込み時のモジュール名で値を確認
789                ←正しく値が変更できている
```

2 回目のモジュール読み込み時には別名 `MyLib1` を与えているが、実体としては同一ライブラリを指し示していることがわかる。

D.1.6 The Zen of Python

`import this` を実行すると、Python 言語についての基本的な考え方や、Python 言語におけるプログラミングに対する姿勢（Tim Peters 氏による）が出力される。

例. The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

D.2 ライブラリ情報の取得

Python 環境で利用するライブラリの管理は、基本的には PSF の `pip` コマンドや Anaconda の `conda` コマンドなどによる²⁷⁸ が、Python のプログラムからライブラリに関する情報（パッケージ情報、モジュール情報）を取得することもできる。

D.2.1 pkg_resources による方法

`setuptools` ライブラリ²⁷⁹ に含まれる `pkg_resources` モジュールを使用すると、当該環境で利用できるライブラリの情報が得られる。このモジュールを用いて利用可能なライブラリの一覧を表示するサンプルプログラムを示す。

プログラム：pkglist.py

```
1 # coding: utf-8
2 import pkg_resources    # ライブラリの読み込み
3 for lib in pkg_resources.working_set:
4     print( lib.project_name, lib.version )
```

解説.

`pkg_resources` ライブラリのオブジェクト `working_set` には当該環境で利用できるライブラリの情報が保持されてい

²⁷⁸付録 A（p.400～）を参照のこと。

²⁷⁹`setuptools` ライブラリがインストールされていない環境もあるので、その場合は手動で（`pip` などによって）これをインストールする必要がある。

る。このプログラムでは `pkg_resources.working_set` から要素（パッケージ情報）を1つずつ取り出して変数 `lib` に受け取り、その `project_name` 属性からライブラリ名を、`version` 属性からバージョン情報を取得している。

このプログラムを実行すると次のように出力される。

実行例.

```
beautifulsoup4 4.12.3
certifi 2024.2.2
chardet 5.2.0
... (途中省略) ...
tzdata 2023.4
tzlocal 5.2
urllib3 2.2.0
```

`pkg_resources` を使用する方法は古いものであり、Python 3.8 以降では次に示す方法（`pkgutil`, `importlib` による方法）が推奨される。

D.2.2 `pkgutil`, `importlib` による方法

`pkgutil` はライブラリ（パッケージ、モジュール）の管理に関する機能を提供するものである。`pkgutil` の `iter_modules` 関数を使用すると、読み込み可能な全てのライブラリの情報を取得することができる。この関数を引数なしで実行すると、個々のライブラリの情報を保持する `ModuleInfo` オブジェクトの並びを表すジェネレータが返される。個々のライブラリの名前は `ModuleInfo` オブジェクトの `name` プロパティから得られる。

例. 読み込み可能なライブラリの名前の一覧表示

```
>>> import pkgutil  [Enter]    ← pkgutil モジュールの読み込み
>>> im = pkgutil.iter_modules() [Enter]    ← ModuleInfo オブジェクトのジェネレータの取得
>>> for m in im:  [Enter]    ← 個々の ModuleInfo オブジェクトの
...     print(m.name) [Enter]    name プロパティを出力
... [Enter]    ← for 文の記述の終了

_asyncio          ← ライブラリ一覧の表示（ここから）
_bz2
_ctypes
    :
    (途中省略)
    :
_pywin
_start_pythonwin
_win32ui
_win32uiOLE
```

`ModuleInfo` オブジェクトの `module_finder` プロパティの `path` プロパティからはそのライブラリが格納されているパスが得られる。上記の例の `for` 文のスイートの中に

```
print(m.module_finder.path)
```

のような記述を追記して試されたい。

`pkgutil` ではライブラリのバージョン情報が得られないので、それら情報を取得するには次に解説する `importlib` を使用する。

`importlib` ライブラリも `pkgutil` と並んでライブラリ（パッケージ、モジュール）の管理に関する機能を提供するものであり、このライブラリの `metadata` モジュールが提供するいくつかの関数（下記）を紹介する。

<code>version(ライブラリ名)</code>	:	ライブラリのバージョン情報を返す。
<code>requires(ライブラリ名)</code>	:	ライブラリが依存する他のライブラリの情報を返す。
<code>files(ライブラリ名)</code>	:	当該ライブラリを構成するファイルのリストを返す。

Python 用の数式処理ライブラリである `sympy` がインストールされている環境で `sympy` の情報を調べる例を次に示す。

例. sympy ライブラリの情報を調べる

```
>>> import importlib.metadata  ←ライブラリの読み込み
>>> importlib.metadata.version('sympy')  ← sympy のバージョン情報の調査
'1.12'
>>> importlib.metadata.requires('sympy')  ← sympy を支えるライブラリの調査
['mpmath (>=0.19)'] ← 0.19 以上の版の mpmath が必要
```

例. sympy ライブラリを構成するファイルを調べる (先の例の続き)

```
>>> fList = importlib.metadata.files('sympy')  ← sympy の構成ファイルリストを取得
>>> for f in fList:  ←個々のファイルを
...     print(f)  表示する
...  ← for 文の記述の終了
../../Scripts/isympy.exe ←構成ファイルの一覧 (ここから)
../../share/man/man1/isympy.1
.._pycache_/_isympy.cpython-311.pyc
isympy.py
sympy-1.12.dist-info/AUTHORS
sympy-1.12.dist-info/INSTALLER
...
(途中省略)
...
sympy/vector/tests/test_vector.py
sympy/vector/vector.py
```

files 関数が返すリストの要素は PackagePath オブジェクトであることに注意すること.

D.3 各種ライブラリの紹介

表 58: 各種ライブラリ

科学技術関連		
ライブラリ名	用途	配布元
matplotlib	グラフ作成／作図	http://matplotlib.org/
NumPy	数値計算（線形代数）	http://www.numpy.org/
SciPy	各種工学のための数値解析	https://www.scipy.org/
SymPy	数式処理	http://www.sympy.org/
データ処理		
ライブラリ名	用途	配布元
scikit-learn	モデリングと機械学習	http://scikit-learn.org/
pandas	表形式データの処理	http://pandas.pydata.org/
seaborn	統計学に適した可視化ツール	https://seaborn.pydata.org/
画像処理関連		
ライブラリ名	用途	配布元
Pillow	画像処理	https://python-pillow.org/
OpenCV	画像処理, カメラキャプチャ, 画像認識	http://opencv.org/
ニューラルネットワーク関連		
ライブラリ名	用途	配布元
PyTorch	深層ニューラルネットワーク	https://pytorch.org/
Keras	深層ニューラルネットワーク	https://keras.io/
マルチメディア／ゲーム関連		
ライブラリ名	用途	配布元
pygame	マルチメディアとゲーム	http://www.pygame.org/
プログラムの高速化／共有ライブラリの呼び出し		
ライブラリ名	用途	配布元
Cython	Python プログラムの高速実行（C 言語変換）	http://cython.org/
Numba	Python プログラムの高速実行（LLVM の応用）	https://numba.pydata.org/
ctypes	共有ライブラリの呼び出し	標準ライブラリ
アプリケーションのビルド		
ライブラリ名	用途	配布元
PyInstaller	単独実行可能なアプリケーションの構築	https://www.pyinstaller.org/

E 対話モードを使いやすくするための工夫

E.1 警告メッセージの抑止と表示

Python 処理系はプログラムの実行中にエラーや例外が発生するとその処理を中断するが、これとは別に**警告メッセージ**（warning）を発行して処理を継続することがある。特に、各種ライブラリを使用する際に警告メッセージが時折表示される。これは、当該ライブラリの開発者の利用者に対する注意喚起などといった意図によるものであり、Python の利用者は警告メッセージを読んでプログラミングにおける判断のための参考にするべきである。

しかし、Python 処理系を対話モードで使用する際、あまりにも多くの警告メッセージが表示されると表示内容の視認性が低下するので、一時的に警告メッセージの表示を抑止した方が良い場面もある。

Python 処理系に標準的に添付されているライブラリ `warnings` を用いると、警告メッセージの表示を抑止する、あるいは独自の警告メッセージを発行することができる。 `warnings` ライブラリを利用する例をプログラム `warnings01.py` に示す。

プログラム： `warnings01.py`

```
1 # coding: utf-8
2 import warnings      # ライブラリの読み込み
3
4 # 警告メッセージの抑止
5 warnings.filterwarnings('ignore')
6 warnings.warn('これは警告メッセージ(a1)です。')
7 warnings.warn('これは警告メッセージ(a2)です。')
8
9 # 警告メッセージの制御を初期状態に戻す
10 warnings.resetwarnings()
11 warnings.warn('これは警告メッセージ(b1)です。')
12 warnings.warn('これは警告メッセージ(b2)です。')
```

このプログラムの 6～7 行目と 11～12 行目は独自の警告メッセージを表示する部分であり、`warn` を実行すると引数に与えた警告メッセージを発行することができる。このプログラムを実行すると次のように表示される。

```
C:\¥warnings01-test.py:6: UserWarning: これは警告メッセージ (a1) です.
warnings.warn('これは警告メッセージ (a1) です. ')
C:\¥warnings01-test.py:7: UserWarning: これは警告メッセージ (a2) です.
warnings.warn('これは警告メッセージ (a2) です. ')
C:\¥warnings01-test.py:11: UserWarning: これは警告メッセージ (b1) です.
warnings.warn('これは警告メッセージ (b1) です. ')
C:\¥warnings01-test.py:12: UserWarning: これは警告メッセージ (b2) です.
warnings.warn('これは警告メッセージ (b2) です. ')
```

プログラムの 5 行目はコメントであるが、先頭の `#` を外すと

```
warnings.filterwarnings('ignore')
```

が有効になり、`filterwarnings` の実行によって以降の警告メッセージが表示されなくなる。（確認されたい）

更に 10 行目先頭の `#` を外すと

```
warnings.resetwarnings()
```

が有効になり、`resetwarnings` の実行により、それ以後は警告メッセージが再び表示されるようになる。（次の実行例を参照のこと）

```
C:\¥warnings01-test.py:11: UserWarning: これは警告メッセージ (b1) です.
warnings.warn('これは警告メッセージ (b1) です. ')
C:\¥warnings01-test.py:12: UserWarning: これは警告メッセージ (b2) です.
warnings.warn('これは警告メッセージ (b2) です. ')
```

※ システムが発行する警告メッセージは必ず読み、その内容を把握すること。 `warnings` に関する詳細は Python の公式インターネットサイトなどを参照のこと。

E.2 メモリの使用状態の管理

Python 処理系は、対話モードをはじめとする対話的なデータ処理環境²⁸⁰ で用いられることが多い。そのような利用形態では、サイズの大きなデータを Python の各種データ構造として次々と読み込んで使用するが、その際、Python 処理系が使用している主記憶上のメモリのサイズには注意を払う必要がある。具体的には、Python 処理系がデータオブジェクトを保持するために使用するメモリの大きさに注意を払う必要がある。

Python 処理系のメモリの使用量が大きくなると、それらを主記憶 (RAM) 上に確保できずに仮想記憶のスワップ²⁸¹ が発生することがあり、これが頻発すると Python 処理系の動作が遅くなる。Python 処理系のメモリの使用を最適なものにするために、以下のような点に注意するべきである。

- Python が使用しているメモリの大きさを適宜調査する
- 不要になったオブジェクトは廃棄する

p.??に示したの応用例 `mstat.py` によって、Python 処理系が使用しているアクティブな物理記憶のサイズと仮想記憶の全サイズを得ることができるが、これとは別に `memory-profiler`²⁸² を使用すると Python 処理系のメモリの使用状況を調べることができる。これは標準ライブラリではないのでインストール作業²⁸³ が必要である。

以下に `memory-profiler` の使用例を示す。

例. `memory-profiler` によるメモリ使用状況の調査

```
>>> from memory_profiler import memory_usage  [Enter] ← memory_profiler の読み込み284
>>> m = memory_usage(proc=-1)  [Enter] ←使用しているメモリサイズを m に取得
>>> print( m )  [Enter] ←値の確認
[22.3515625] ←約 21MiB 使用している
```

これは、Python 処理系を起動した直後に `memory-profiler` を読み込み、`memory_usage` を実行して使用メモリサイズを調べた例である。次に、非常に大きなサイズのリストを作成し、その直後の使用メモリサイズを調べる。

例. 大きなデータを作成した直後のメモリ使用状況 (先の例の続き)

```
>>> a = [float(x) for x in range(10**8)]  [Enter] ←要素数が 108 個のリストを作成
>>> m = memory_usage(proc=-1)  [Enter] ←使用しているメモリサイズを m に取得
>>> print( m )  [Enter] ←値の確認
[3849.20703125] ←約 3.75GiB 程使用している
```

a に要素数が 10⁸ 個のリストが作成されており、Python 処理系のメモリの使用量が約 3.5GB に増加したことがわかる。この a を `del` 文により削除するとメモリの使用量が削減される。(次の例参照)

例. 大きなデータを削除した直後のメモリ使用状況 (先の例の続き)

```
>>> del a  [Enter] ←巨大なリスト a を削除
>>> m = memory_usage(proc=-1)  [Enter] ←使用しているメモリサイズを m に取得
>>> print( m )  [Enter] ←値の確認
[24.26953125] ←約 24MiB 程度になった
```

`memory-profiler` に関する詳細は当該ソフトウェアの公式インターネットサイトなどの情報を参照のこと。

E.2.1 オブジェクトのサイズの調査

`sys` モジュールの `getsizeof` を用いると、個々のオブジェクトのサイズを調べることができる。

²⁸⁰IPython, JupyterLab などを含む。これらに関しては他のドキュメントを参照のこと。

²⁸¹主記憶と補助記憶の間でのデータの交換。

²⁸²公式インターネットサイト：<https://pypi.org/project/memory-profiler/>

²⁸³「A.4 PIP によるライブラリ管理」(p.404) を参照のこと。

²⁸⁴ライブラリの名称は `memory-profiler` であるが、読み込み時は `memory_profiler` とアンダースコアを用いた表記であることに注意すること。

例. 変数に与えられているオブジェクトのサイズを調べる

```
>>> import sys  ←モジュールの読み込み
>>> a = 123  ←変数 a に整数型オブジェクトを割り当てる
>>> sys.getsizeof( a )  ←そのサイズを調べる
28      ←変数 a が持つオブジェクトのサイズ：28 バイト
```

この例のように、getsizeof は引数に与えられたオブジェクトのサイズをバイト単位で返す。

▲注意▲

getsizeof は調査対象のオブジェクトが参照しているオブジェクトは調査対象としない。(次の例参照)

例. オブジェクト全体のサイズが得られないケース (先の例の続き)

```
>>> lst = [[float(n+m*100) for n in range(100)] for m in range(100)]  ←大きなリスト
>>> sys.getsizeof( lst )  ←そのサイズを調べる
920      ←全要素を含めたサイズにはなっていない
```

オブジェクト全体のサイズを調べるには、当該オブジェクトが参照しているオブジェクトに関しても再帰的にサイズを調べて、それらを合計する必要がある。このための方法に関しては Python の公式インターネットサイトの `sys.getsizeof` の項に例が紹介されている。実際のソースコードも GitHub の下記の URL で公開されている。

<https://github.com/ActiveState/recipe-577504-compute-mem-footprint/blob/master/recipe.py>

このソースコード 'recipe.py' の中に定義されている関数 `total_size` を読み込んで上の例のリストのサイズを調べる例を示す。

例. `total_size` の実行例

```
>>> from recipe import total_size  ←ソースコードをモジュールとして読み込む
>>> total_size( lst )  ←リストのサイズを調べる
332920      ← 300 キロバイト以上ある
```

F 文書化文字列と関数アノテーション

Python には `help` 関数があり、関数やクラスの説明を出力することができる。次の例は、Python の組み込み関数 `print` に関する説明文を表示するものである。

例. `print` 関数の説明を表示する

```
>>> help(print)  [Enter]    ← help 関数による説明文の表示
Help on built-in function print in module builtins:    ←ここから説明文の表示
print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.
    :
    (以下省略)
    :
```

プログラマが作成した関数やクラスにも説明文を与えることができる。例えば、引数に与えた値を 2 倍する関数 `dbl` を次のように定義する。

例. 値を 2 倍する関数 `dbl` の定義

```
>>> def dbl(n):  [Enter]    ←関数 dbl の定義
...     return 2*n  [Enter]
...  [Enter]    ←関数定義の記述の終了
>>> dbl(3)  [Enter]    ←関数 dbl の評価（実行）
6           ←戻り値
```

この関数には未だ説明文が与えられておらず、`help` 関数で説明を求めると次のようになる。

例. 関数 `dbl` の説明を求める試み（先の例の続き）

```
>>> help(dbl)  [Enter]    ← help 関数による説明文の表示を試みると…
Help on function dbl in module __main__:
dbl(n)          ←出力される説明はこれだけ
```

関数 `dbl` に「`n` を 2 倍する関数」という説明文を与えるには次のような形で関数定義を行う。

例. 値を 2 倍する関数 `dbl` の定義（説明機能つき）

```
>>> def dbl(n):  [Enter]    ←関数 dbl の定義
...     'n を 2 倍する関数'  [Enter]    ←定義内容の先頭に説明文の文字列を配置する
...     return 2*n  [Enter]
...  [Enter]    ←関数定義の記述の終了
>>> dbl(3)  [Enter]    ←関数 dbl の評価（実行）
6           ←戻り値
>>> help(dbl)  [Enter]    ← help 関数による説明文の表示
Help on function dbl in module __main__:
dbl(n)
    n を 2 倍する関数    ←説明文が表示されている
```

このように、関数の定義内容の先頭に説明文の文字列（**文書化文字列**：docstring）を記述すると、`help` による説明表示の際にそれが出力される。

文書化文字列は複数行に渡ることが一般的であり、そのような説明文を記述する際は「`'''～'''`」の形式（3 重引用符）で文字列を記述する。

文書化文字列は、その関数の `__doc__` プロパティに保持されている。

例. `__doc__` プロパティの参照（先の例の続き）

```
>>> dbl.__doc__  [Enter]    ←文書化文字列を参照
'n を 2 倍する関数'        ←結果
```

F.1 関数アノテーション (Function Annotations)

関数定義を記述する際、仮引数と戻り値にアノテーション（注釈）を付けることができ、help による説明表示の際にそれを出力することができる。

例. 関数アノテーション付きの定義

```
>>> def dbl( n:'数値' ) -> '数値':  ←仮引数と戻り値にアノテーションを付けている
...     'n を 2 倍する関数' 
...     return 2*n 
...  ←関数定義の記述の終了
>>> help(dbl)  ← help 関数による説明文の表示
Help on function dbl in module __main__:
dbl(n: '数値') -> '数値'      ←アノテーションが表示されている
    n を 2 倍する関数        ←説明文が表示されている
```

このように、各仮引数の後ろにコロン「:」を記述し、続けて説明文の文字列を記述する。戻り値に関するアノテーションは、仮引数の後ろの閉じ括弧「)」の次に「->」を記述し、続けて説明文の文字列を記述する。

関数アノテーションは、その関数の `__annotations__` プロパティに保持されている。

例. `__annotations__` プロパティの参照（先の例の続き）

```
>>> dbl.__annotations__  ←関数アノテーションを参照
{'n': '数値', 'return': '数値'}      ←辞書の形で得られる
```

このように、`__annotations__` から辞書の形（キーは仮引数の名前、戻り値のキーは 'return'）で関数アノテーションが得られる。

クラス定義に文書化文字列を与える場合は `class` 文の直後にそれを記述する。また、メソッドに対しては、関数定義の場合に準じた形で文書化文字列と関数アノテーションを与える。

参考.

Python に標準添付の `typing` モジュールを用いて関数アノテーションを記述することが推奨されている。ただし本書ではこれに関しては言及しない。（必要な場合は公式インターネットサイトなどを参照のこと）

G 型ヒント

Python は動的な型付けの言語であり、変数に代入することができる値の型に制限はない。また、関数やメソッドの引数についても同様である。このことは次のサンプルプログラム `typing00.py` でも確認することができる。

プログラム：`typing00.py`

```
1 def f(x,y):
2     '''returns x+y'''
3     return x+y
4
5 if __name__ == '__main__':
6     print( '1 + 2 ->', f(1,2) )
7     print( '1.0 + 2.0 ->', f(1.0,2.0) )
8     print( '"1" + "2" ->', f('1','2') )
```

このプログラムをスクリプトとして Python 言語処理系で実行すると標準出力に次のような出力が得られる。

```
1 + 2 -> 3
1.0 + 2.0 -> 3.0
"1" + "2" -> 12
```

関数 `f` が引数に `int`, `float`, `str` それぞれの型の値を受け取って処理している様子がわかる。ここで留意すべき点として、関数 `f` の目的が数値の加算であるのか、文字列の連結であるのかが曖昧なことがある。

Python 言語では、関数やメソッドの引数、あるいは通常の変数に対して**型ヒント**を与えることができる。具体的には、関数やメソッドの仮引数の直後にコロンの「:」を記述して、その後に受け取るべきデータの型を記述する。変数に値を代入する際も同様の形式で型ヒントを与えることができる。

先のプログラムの関数 `f` に型ヒントを付けた例が次の `typing01.py` である。

プログラム：`typing01.py`

```
1 def f( x:int, y:int ) -> int:
2     '''returns x+y'''
3     return x+y
4
5 if __name__ == '__main__':
6     print( '1 + 2 ->', f(1,2) )
7     print( '1.0 + 2.0 ->', f(1.0,2.0) )
8     print( '"1" + "2" ->', f('1','2') )
```

このプログラムからわかるように、関数定義の冒頭で「-> 型名」を付けることで当該関数の戻り値に型ヒントを付けることができる。このプログラムを実行すると `typing00.py` と同様の出力が得られる。

G.1 型ヒントの存在意義

上の2つの例 `typing00.py`, `typing01.py` からわかるように、型ヒント自体はプログラムの実行に関してもいかなる制限も与えないが、規模の大きなプログラムを多人数で開発する状況では様々な利便性をもたらす。その利便性の1つが**ドキュメント生成における補助**（help 機能の補助）である。例えば、型ヒントの無い `typing00.py` の関数 `f` のドキュメントを対話環境で出力する例を次に示す。

例. `typing00.py` の関数 `f` のドキュメント表示

```
>>> import typing00 Enter ←モジュールとしてプログラムを読み込む
>>> help(typing00.f) Enter ←関数 f のドキュメント表示
Help on function f in module typing00:

f(x, y)
    returns x+y
```

このように、関数の引数と戻り値の型に関する説明は出力されない。ただし、`typing01.py` のように型ヒントの記述があると、それが関数のドキュメントとして出力される。（次の例）

例. typing01.py の関数 f のドキュメント表示

```
>>> import typing01  ←モジュールとしてプログラムを読み込む
>>> help(typing01.f)  ←関数 f のドキュメント表示
Help on function f in module typing01:
f(x: int, y: int) -> int      ←引数と戻り値の型ヒントが表示されている
    returns x+y
```

型ヒントは関数の実装目的を明確にし、プログラムの可読性の向上に役立つ、例えば、上の typing01.py の場合、関数 f の実装目的は「与えられた 2 つの整数の和を求める」機能の実現であることが理解できる。これに対して、最初の typing00.py では、関数 f が数値の和を求めるためのものか、文字列を連結するためのものかが理解し難い。

プログラムの実装目的の不明瞭さは、それを応用する者にとって予期しない結果をもたらすリスクを高めることにつながり、規模の大きなシステムを多人数で開発する際に好ましくない。実際に、型ヒントに基づいてプログラムを診断する mypy などのツールも存在しており、プログラム開発の現場で用いられている。また、プログラム開発を支援する各種のツール²⁸⁵ も、型ヒントに基づく補助機能を提供するものが多い。

G.2 mypy によるプログラムの静的診断

mypy は Python プログラムの型を静的に診断するツールであり、Jukka Lehtosalo 氏によって開発されて公開²⁸⁶されている。mypy はその利用に先立って、予め pip などシステムに導入しておく必要がある。mypy は OS のコマンドとして次のようにして起動する。

py -m mypy	診断対象のソースファイル	Windows 版の PSF の Python の場合
python -m mypy	診断対象のソースファイル	それ以外の Python の場合
mypy	診断対象のソースファイル	mypy がコマンドとして認識される場合

先の typing01.py を mypy で診断する例を次に示す。

例. mypy による typing01.py の診断

```
C:\Users\katsu>py -m mypy typing01.py  ←診断の開始
typing01.py:7: error: Argument 1 to "f" has incompatible type "float"; expected "int" [arg-type]
typing01.py:7: error: Argument 2 to "f" has incompatible type "float"; expected "int" [arg-type]
typing01.py:8: error: Argument 1 to "f" has incompatible type "str"; expected "int" [arg-type]
typing01.py:8: error: Argument 2 to "f" has incompatible type "str"; expected "int" [arg-type]
Found 4 errors in 1 file (checked 1 source file)
```

このように、関数 f の各引数が整数でない旨の診断結果が出力される。

G.3 変数に対する型ヒント

型ヒントは通常の変数（グローバル、ローカル）にも適用可能である。このことを次のサンプル typing04.py で確認する。

プログラム：typing04.py

```
1 x:int = 3
2 s:int = '文字列'
3 print(f'{x=}, {s=}')
```

このプログラムでは変数 s の型ヒントが int となっているが、文字列が代入されている。このプログラムは実行時にエラーは発生しないが、これを mypy で診断する例を次に示す。

例. mypy による typing04.py の診断

```
C:\Users\katsu>py -m mypy typing04.py  ←診断の開始
typing04.py:2: error: Incompatible types in assignment (expression has type "str", variable has type "int") [assignment]
Found 1 error in 1 file (checked 1 source file)
```

²⁸⁵Microsoft 社の Visual Studio Code や JetBrains 社の PyCharm、IPython 環境のノートブックも型ヒントに基づくコーディング作業の補助機能を提供している。

²⁸⁶<https://github.com/python/mypy>

プログラムの2行目で、型ヒントに違反する代入処理が行われていることが報告されている。

G.4 型ヒントのためのライブラリ： typing モジュール

型ヒントのための様々な機能を提供する標準ライブラリとして typing モジュールがある。例えば先の例における関数 f の目的を「数値の加算を行うものである」とするならば、整数 (int)、浮動小数点数 (float) の両方を対象とする形で型ヒントを与えるべきである。そのような場合に、typing モジュールの Union が役立つ。

書き方： Union[型 1, 型 2, ...]

「型 1」, 「型 2」, ... の全てに対応する型ヒントを与える。これの応用例を次のサンプル typing02.py に示す。

プログラム：typing02.py

```
1 from typing import Union
2
3 def f( x:Union[int,float], y:Union[int,float] ) -> Union[int,float]:
4     '''returns x+y'''
5     return x+y
6
7 if __name__ == '__main__':
8     print( '1 + 2 ->', f(1,2) )
9     print( '1.0 + 2.0 ->', f(1.0,2.0) )
10    print( '"1" + "2" ->', f('1','2') )
```

このプログラムでは、関数 f を整数と浮動小数点数に対して加算を実行するものとして定義していることが明確になる。このプログラムを mypy で診断した例を次に示す。

例. mypy による typing02.py の診断

```
C:\Users\katsu>py -m mypy typing02.py [Enter] ←診断の開始
typing02.py:10: error: Argument 1 to "f" has incompatible type "str"; expected "int | float" [arg-type]
typing02.py:10: error: Argument 2 to "f" has incompatible type "str"; expected "int | float" [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

関数 f の引数に文字列を与えた場合のみ指摘していることがわかる。typing02.py の関数 f のドキュメントを対話環境で出力する例を次に示す。

例. typing02.py の関数 f のドキュメント表示

```
>>> import typing02 [Enter] ←モジュールとしてプログラムを読み込む
>>> help(typing02.f) [Enter] ←関数 f のドキュメント表示
Help on function f in module typing02:
f(x: Union[int, float], y: Union[int, float]) -> Union[int, float] ←型ヒントの表示
    returns x+y
```

Union と同様の型ヒントを演算子「|」で記述することもでき²⁸⁷、typing02.py と同等の型ヒントの記述を typing02-2.py のようにすることもできる。

プログラム：typing02-2.py

```
1 def f( x:int|float, y:int|float ) -> int|float:
2     '''returns x+y'''
3     return x+y
4
5 if __name__ == '__main__':
6     print( '1 + 2 ->', f(1,2) )
7     print( '1.0 + 2.0 ->', f(1.0,2.0) )
8     print( '"1" + "2" ->', f('1','2') )
```

型ヒントに基づく静的診断でプログラムのエラーを事前に確認できる場合がある。次の typing03.py における関数 g は引数に与えられた数値の差を求めるものであり、「-」演算子が使えないデータを受け取るとエラーとなる。

²⁸⁷Python 3.10 から適用。

プログラム：typing03.py（誤りを含んだプログラム）

```
1 def g( x:int|float, y:int|float ) -> int|float:
2     '''returns x-y'''
3     return x-y
4
5 if __name__ == '__main__':
6     print( '1 - 2 ->', g(1,2) )
7     print( '1.0 - 2.0 ->', g(1.0,2.0) )
8     print( '"1" - "2" ->', g('1','2') )
```

このプログラムを mypy で診断すると次のようになる。

例. mypy による typing03.py の診断

```
C:\Users\katsu>py -m mypy typing03.py  ←診断の開始
typing03.py:8: error: Argument 1 to "g" has incompatible type "str"; expected "int | float" [arg-type]
typing03.py:8: error: Argument 2 to "g" has incompatible type "str"; expected "int | float" [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

プログラムの 8 行目に不適切な型のデータ（文字列）が関数 g に与えられていることが報告されていることがわかる。実際にこのプログラムをスクリプトとして Python 言語処理系に与えると次のようにエラーが発生する。

例. typing03.py 実行におけるエラー

```
C:\Users\katsu>py typing03.py  ←実行
1 - 2 -> -1
1.0 - 2.0 -> -1.0
Traceback (most recent call last):          ←エラーが起こる
  File "C:\Users\katsu\typing03.py", line 8, in <module>
    print( '"1" - "2" ->', g('1','2') )
    ~~~~~~
  File "C:\Users\katsu\typing03.py", line 3, in g
    return x-y
    ~~~
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

実際に、プログラムの 8 行目（と 3 行目）でエラーが発生している。

G.4.1 様々な型ヒント

typing モジュールが提供する型ヒントには様々なものがある。

データ構造でない基本型²⁸⁸の型ヒントを表 59 に示す。

表 59: 基本型の型ヒント（一部）

型ヒント	解 説	型ヒント	解 説
int	整数	float	浮動小数点数
complex	複素数	bool	真理値
str	文字列	bytes	バイト列
None	None を表す（NoneType ではない）		

様々な型ヒントを応用した特殊な型ヒントを表 60 に示す。

代表的なデータ構造についての型ヒントを表 61 に示す。

表 60, 61 の型ヒントを使う際は、次のようにして import すると便利である。

例. List, Union の読み込み

```
from typing import List, Union
```

²⁸⁸組み込みのデータ型の内、スカラーと呼ばれるデータの型。

表 60: 特殊な型ヒント (一部)

型ヒント	解 説
<code>Union[T1,T2,...]</code>	型 <code>T1,T2,...</code> のどれか
<code>Literal[V1,V2,...]</code>	値 <code>V1,V2,...</code> のどれか (異なる型の値を並べても良い)
<code>Optional[T]</code>	型 <code>T</code> か <code>None</code>
<code>Type[C]</code>	クラス名 <code>C</code> を型ヒントにする
<code>Callable[[A1,A2,...,An],R]</code>	関数オブジェクトの型ヒント. <code>A1,A2,...,An</code> は引数の型ヒント, <code>R</code> は戻り値の型ヒント.
<code>Any</code>	任意の型. (あまり使わない方が良い)

表 61: データ構造の型ヒント (一部)

型ヒント	解 説
<code>List[T]</code>	型ヒント <code>T</code> の要素が並ぶリスト. 複数の型の要素を許容する場合は <code>T</code> に表 60 の型ヒントを適用する.
<code>Tuple[T1,T2, ..., Tn]</code> <code>Tuple[T, ...]</code>	型ヒント <code>T1,T2,...Tn</code> の要素が並ぶタプル (<code>n</code> 個の固定長) 型ヒント <code>T</code> の要素が並ぶタプル. この場合の <code>'...'</code> は Ellipsis. 複数の型の要素を許容する場合は <code>T</code> に表 60 の型ヒントを適用する.
<code>Set[T]</code>	型ヒント <code>T</code> の要素のセット. 複数の型の要素を許容する場合は <code>T</code> に表 60 の型ヒントを適用する. また <code>T</code> はイミュータブルなものに限る.
<code>Dict[KT,VT]</code>	型ヒントが <code>KT</code> のキー, 型ヒントが <code>VT</code> の値から成るエントリを持つ辞書オブジェクト. <code>KT, VT</code> には様々な型ヒントが指定できるが, <code>KT</code> はイミュータブルなものに限る.

H スロットベースのクラス

Python のクラス定義では, クラス変数 `__slots__` の設定によって, そのインスタンスが保持できるインスタンス変数を限定することができる.

例. スロットベースのクラス `SC` の定義

```
>>> class SC: Enter      ←クラス定義の記述の開始
...     __slots__ = ( 'a', 'b' ) Enter  ←使用可能なインスタンス変数の記述
...     def __init__(self, v1, v2): Enter
...         self.a = v1 Enter
...         self.b = v2 Enter
...     def show(self): Enter
...         print( f'{self.a=}, {self.b=}' ) Enter
... Enter      ←クラス定義の記述の終了
```

このクラス `SC` ではクラス変数 `__slots__` が設定されている. この例のように, タプルなどのイテラブルの形式で, 使用できるインスタンス変数の名前を文字列で指定する. このような形で定義されるクラスを本書では**スロットベースのクラス**と呼ぶ.

スロットベースのクラスにおいても, インスタンスの生成やメソッドの実行は通常のクラスの場合と同様である. (次の例)

例. インスタンスの生成とメソッドの実行 (先の例の続き)

```
>>> m = SC( 10, 20 ) Enter      ←インスタンスの生成
>>> m.show() Enter      ←メソッドの実行
self.a=10, self.b=20      ←実行結果
```

スロットベースのクラスのインスタンスは `__dict__` 属性を持たず, 新たなインスタンス変数を追加することができない. (次の例)

例. 新たなインスタンス変数を追加する試み（先の例の続き）

```
>>> m.c = 30  [Enter]    ←新たなインスタンス変数 c を追加しようとする
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
    m.c = 30
    ~~~
AttributeError: 'SC' object has no attribute 'c' and no __dict__ for setting new attributes
```

また、スロットベースのインスタンスは `__dict__` 属性を持たないことから、`vars` 関数も使えない。（次の例）

例. `vars` 関数実行の試み（先の例の続き）

```
>>> vars(m)  [Enter]    ← vars 関数を実行しようとする
Traceback (most recent call last):    ←エラーとなる
  File "<stdin>", line 1, in <module>
    vars(m)
    ~~~~~
TypeError: vars() argument must have __dict__ attribute
```

H.1 スロットベースのクラスが役立つケース

スロットベースのクラスのインスタンスは `__dict__` 属性を持たないことから、記憶資源の使用量が少ない。従って、主記憶の使用可能なサイズに厳しい制限がある状況ではこの形式のクラスが役立つことがある。

先の例で作成したインスタンス `m` と、通常のクラス（`__slots__` を持たないクラス）のインスタンスの間でサイズの大きさを比較する例を次に挙げる。

例. 通常のクラス（`__slots__` を持たないクラス）の定義（先の例の続き）

```
>>> class NC:  [Enter]    ←クラス定義の記述の開始
...     def __init__(self, v1, v2):  [Enter]
...         self.a = v1  [Enter]
...         self.b = v2  [Enter]
...     def show(self):  [Enter]
...         print( f'{self.a=}, {self.b=}' )  [Enter]
...  [Enter]    ←クラス定義の記述の終了
>>> n = NC(10,20)  [Enter]    ←インスタンスの生成
>>> n.show()  [Enter]    ←メソッドの実行
self.a=10, self.b=20    ←実行結果
```

この例で定義されたクラス `NC` はクラス変数 `__slots__` を持たないので、そのインスタンス `n` は `__dict__` 属性を持つ。次の例は、インスタンス `n` と `m` の主記憶上のサイズを求めるものである。

例. 上記 `n` と `m` のサイズの調査（先の例の続き）

```
>>> import sys  [Enter]
>>> sys.getsizeof(n) + sys.getsizeof(n.__dict__)  [Enter]    ← n のサイズの調査
344    ← 344 バイト
>>> sys.getsizeof(m)  [Enter]    ← m のサイズの調査
48    ← 48 バイト
```

この例から、スロットベースのクラスのインスタンスの方が記憶資源の使用量が少ないことがわかる。

H.2 スロットベースのクラスを使用する上での注意点

クラスの継承（クラスの拡張）に関しては「スロットベース」である性質は引き継がれない。従って、サブクラス（拡張クラス）をスロットベースにする際は、サブクラス側でも明示的に `__slots__` を設置しなければならないので注意が必要である。このことを以下に例示する。

例. スロットベースのクラスを拡張する試み（先の例の続き）

```
>>> class SubSC(SC): pass      ←先のスロットベースのクラスを拡張
...                               ←クラス定義の記述の終了
>>> SubSC.__slots__            ←クラス変数 __slots__ の確認
('a', 'b')                    ←ただしこれは、基底クラス（親クラス）側のものを継承している
```

サブクラス SubSC 側にも __slots__ が継承されているので、このサブクラスもスロットベースのクラスになっているような印象を与えるが、次の例から、そうでないことがわかる。

例. サブクラスのインスタンス生成（先の例の続き）

```
>>> msub = SubSC(10,20)        ←インスタンスを生成
>>> msub.show()                ←メソッドが継承されており
                                self.a=10, self.b=20          ←正しく動作するが
>>> msub.__dict__              ←__dict__ 属性が
{}                              ←作成されており
>>> msub.c = 30                ←新たな属性を追加登録することができる
>>> vars(msub)                  ←しかも新たに登録した属性のみが
{'c': 30}                       ← __dict__ に登録されている
```

このことは Python 利用者を困惑させる原因となる。

結論として、スロットベースのクラスを定義するには、当該クラスで明示的に __slots__ を設置しなければならない。（次の例）

例. サブクラス側もスロットベースにする方法（先の例の続き）

```
>>> class SubSC2(SC):          ←サブクラス側もスロットベースにするには
...     __slots__ = ( 'a', 'b', 'c' )  ←明示的に __slots__ を設置する
...                                     ←クラス定義の記述の終了
>>> msub2 = SubSC2(10,20)      ←インスタンスを生成
>>> msub2.show()               ←メソッドが継承されており
                                self.a=10, self.b=20          ←正しく動作し
>>> msub2.__dict__             ← __dict__ も作成されず、参照を試みると
Traceback (most recent call last):      ←エラーとなる
  File "<stdin>", line 1, in <module>
    msub2.__dict__
AttributeError: 'SubSC2' object has no attribute '__dict__'. Did you mean: '__dir__'?
```

■ 通常のクラスを継承したクラス（サブクラス）側に __slots__ を設置した場合の挙動

通常のクラス（__slots__を持たないクラス）のサブクラス（拡張クラス）に __slots__ を設置しても、そのクラスのインスタンスに __dict__ ができてしまい、属性を __slots__ によって制限できなくなるので注意が必要である。このことについて例を挙げて説明する。

例. 通常のクラスの定義

```
>>> class NC:                  ←クラス定義の記述の開始
...     def __init__(self,v):  ←
...         self.a = v        ←
...     def show(self):        ←
...         print( f'{self.a=}' )  ←
...                               ←クラス定義の記述の終了
>>> n = NC(99)                ←インスタンスの生成と
>>> n.show()                  ←メソッドの実行
self.a=99
```

次に、このクラスを拡張したクラス（サブクラス） SC を定義する。

例. サブクラス側に`__slots__`を設置する（先の例の続き）

```
>>> class SC(NC): Enter    ←先のクラスを継承（拡張）して
...     __slots__ = ('x',) Enter    ←__slots__を設置する
... Enter    ←クラス定義の記述の終了
>>> m = SC(88) Enter    ←インスタンスの生成
>>> m.x = 77 Enter    ←__slots__に指定した属性に値を設定して
>>> m.x Enter    ←参照することが
77                                ←できる
>>> m.__dict__ Enter    ←しかし、インスタンス生成時に与えた初期値は
{'a': 88}                        ←__dict__で管理される
```

この例からわかるように、スーパークラス（基底クラス）NC の性質をサブクラス（拡張クラス）SC が継承しており、`__slots__` による使用可能属性の限定ができない。

■ まとめ

- ・ システムの開発において、スロットベースのクラス定義は優先的に採用すべきでない。
- ・ スロットベースのクラス定義は、クラス間の継承関係の中で行うべきでない。

スロットベースのクラスは、Python の標準ライブラリを構成する際の最適化に用いられることが多く、一般的な Python 言語の利用者にはその使用は推奨されていない。

スロットベースのクラスは、Python 元来のオブジェクト指向プログラミングの考え方に沿っていない部分が多く、また、スロットベースのクラスを使用しても、プログラムの実行速度の面ではあまり改善が見られない。従って、多量のオブジェクトを作成して使用するケースにおいても、NumPy²⁸⁹ などの高速処理のための配列を応用して工夫する方が賢明である。

²⁸⁹ サードパーティが開発する高速な数値演算ライブラリ。NumPy については拙書「Python3 ライブラリブック」で解説しています。

I サンプルプログラム

I.1 リスト／セット／辞書のアクセス速度の比較

I.1.1 スライスに整数のインデックスを与える形のアクセス

「 n 番目の要素」という形で要素にアクセスする場合の速度比較を行うプログラムが `spdTest00.py` である。このプログラムは長い (10^6 個の要素を持つ) リスト `L`, 辞書 `D` を作成し, それらの n 番目の要素にランダムにアクセスするものである。またランダムなアクセスを, その要素の個数と同じ回数 (10^6 回) リスト, 辞書それぞれに対して行い, 実行時間を計測する。

プログラム: `spdTest00.py`

```
1  # coding: utf-8
2  import time
3  import secrets
4  #####
5  # 実行速度テスト (インデックスによる) #
6  #####
7  #--- インデックスのランダムアクセス ---
8  def spdTestIdx( Data ):
9      n = len( Data )
10     t1 = time.time()
11     for c in range(n):
12         I = secrets.randbelow(n)
13         Data[I] = I
14     t2 = time.time()
15     return t2 - t1
16
17 def spdTestIdxAvr( Data, n ):
18     tL = []
19     for c in range(n):
20         t = spdTestIdx( Data )
21         print( c+1, '回目:\t', t, '秒' )
22         tL.append( t )
23     avr = sum( tL ) / n
24     print( '平均:\t', avr, '秒' )
25     return avr
26
27 #--- 実行時間テスト ---
28 N = 1000000          # データサイズ
29 L = list( range(N) ) # リスト
30 D = { x:x for x in range(N) } # 辞書
31
32 print( 'リストの場合のテスト' )
33 print( '-----' )
34 aL = spdTestIdxAvr( L, 3 )
35
36 print( '\n辞書の場合のテスト' )
37 print( '-----' )
38 aD = spdTestIdxAvr( D, 3 )
39 print( 'リストの場合に対する速度比:\t', aL/aD, '倍' )
```

このプログラムの実行例を次に示す。

例. spdTest00.py の実行例

C:\Users\katsu>py spdTest00.py Enter ←コマンドからスクリプトを起動

リストの場合のテスト

```
-----
1 回目:  0.5842320919036865 秒
2 回目:  0.5850205421447754 秒
3 回目:  0.57867431640625 秒
平均:    0.5826423168182373 秒
```

辞書の場合のテスト

```
-----
1 回目:  0.7023506164550781 秒
2 回目:  0.7070534229278564 秒
3 回目:  0.7088878154754639 秒
平均:    0.7060972849527994 秒
リストの場合に対する速度比:      0.8251586987155534 倍
```

※ 実行環境: Python 3.11.2, Intel Corei7-1195G7 2.9GHz, 32GB RAM, Windows11 Pro

(評価)

リストの方が辞書に比べて若干早いことがわかる。

I.1.2 メンバシップ検査に要する時間

データ構造の中に特定の要素があるかどうかを調べるのに要する時間を調べるプログラムが spdTest01.py である。このプログラムでは 30,000 個の要素を持つデータ構造に対して要素の含有検査（メンバシップ検査）を行う。リスト L, セット S, 辞書 D はそれぞれ 0~29,999 の整数を要素として持ち、発生した整数の乱数がそのデータ構造に含まれるかどうかを検査する。メンバシップ検査は要素の個数と同じ回数繰り返して、その実行に要した時間を計測する。

プログラム: spdTest01.py

```
1  # coding: utf-8
2  import time
3  import secrets
4  #####
5  # 実行速度テスト(1)                                     #
6  #####
7  #--- メンバシップ検査の速度テスト ---
8  def spdTest( Data ):
9      n = len( Data )
10     t1 = time.time()
11     for c in range(n):
12         chk = secrets.randbelow(n) in Data
13     t2 = time.time()
14     return t2 - t1
15
16 def spdTestAvr( Data, n ):
17     tL = []
18     for c in range(n):
19         t = spdTest( Data )
20         print( c+1, '回目:\t', t, '秒' )
21         tL.append( t )
22     avr = sum( tL ) / n
23     print( '平均:\t', avr, '秒' )
24     return avr
25
26 #--- 実行時間テスト ---
27 N = 30000                                     # 要素の個数
28 L = list( range(N) )                         # リスト
29 S = set( L )                                 # セット
30 D = { x:x for x in range(N) }               # 辞書
31 i = 3                                         # 検査実行回数
32
33 print( 'リストの場合のテスト' )
34 print( '-----' )
35 aL = spdTestAvr( L, i )
36
```

```

37 | print( '\nセットの場合のテスト' )
38 | print( '-----' )
39 | aS = spdTestAvr( S, i )
40 | print( 'リストの場合に対する速度比:\t', aL/aS, '倍' )
41 |
42 | print( '\n辞書の場合のテスト' )
43 | print( '-----' )
44 | aD = spdTestAvr( D, i )
45 | print( 'リストの場合に対する速度比:\t', aL/aD, '倍' )

```

このプログラムの実行例を次に示す。

例. spdTest01.py の実行例

C:\¥Users¥katsu>py spdTest01.py Enter ←コマンドからスクリプトを起動

リストの場合のテスト

```

1 回目:  1.7754509449005127 秒
2 回目:  1.777921199798584 秒
3 回目:  1.7940375804901123 秒
平均:    1.782469908396403 秒

```

セットの場合のテスト

```

1 回目:  0.015633106231689453 秒
2 回目:  0.015620708465576172 秒
3 回目:  0.015627622604370117 秒
平均:    0.015627145767211914 秒
リストの場合に対する速度比:      114.06241004907571 倍

```

辞書の場合のテスト

```

1 回目:  0.019140958786010742 秒
2 回目:  0.012587308883666992 秒
3 回目:  0.015713214874267578 秒
平均:    0.015813827514648438 秒
リストの場合に対する速度比:      112.71590680657742 倍

```

※ 実行環境：Python 3.11.2, Intel Corei7-1195G7 2.9GHz, 32GB RAM, Windows11 Pro

(評価)

セットと辞書は共に同等の実行速度であり、リストに比べて100 倍以上早いことがわかる。

次に、spdTest01.py で調べた実行時間が、データの個数が増えるのに対してどのように伸びてゆくかを調べる。次のサンプルプログラム spdTest02.py で調べる。(グラフ描画に matplotlib を要する)

プログラム：spdTest02.py

```

1 | # coding: utf-8
2 | import time
3 | import secrets
4 | import matplotlib.pyplot as plt
5 | #####
6 | # 実行速度テスト(2)                                     #
7 | #####
8 | #--- メンバシップ検査の速度テスト ---
9 | def spdTest( Data ):
10 |     n = len( Data )
11 |     t1 = time.time()
12 |     for c in range(n):
13 |         chk = secrets.randbelow(n) in Data
14 |     t2 = time.time()
15 |     return t2 - t1
16 |
17 | def spdTestAvr( Data, n ):
18 |     tL = []
19 |     for c in range(n):
20 |         t = spdTest( Data )
21 |         tL.append( t )

```

```

22     avr = sum( tL ) / n
23     return avr
24
25 #--- 実行時間テスト ---
26 N = 10000                                # 要素の個数
27 X = range(0,N,100)
28 A1 = [];    A2 = [];    A3 = []          # 実行時間のリスト
29 for n in X:
30     D = list( range(n) )
31     A1.append( spdTestAvr( D, 3 ) )
32     D = set( D )
33     A2.append( spdTestAvr( D, 3 ) )
34     D = { x:x for x in range(n) }
35     A3.append( spdTestAvr( D, 3 ) )
36
37 plt.plot(list(X),A1,label='List')
38 plt.plot(list(X),A2,label='Set')
39 plt.plot(list(X),A3,label='Dict')
40 plt.legend()
41 plt.title('test for List/Set/Dict')
42 plt.show()

```

このプログラムの実行結果の例を図 74 に示す。(実行環境は先と同じ)

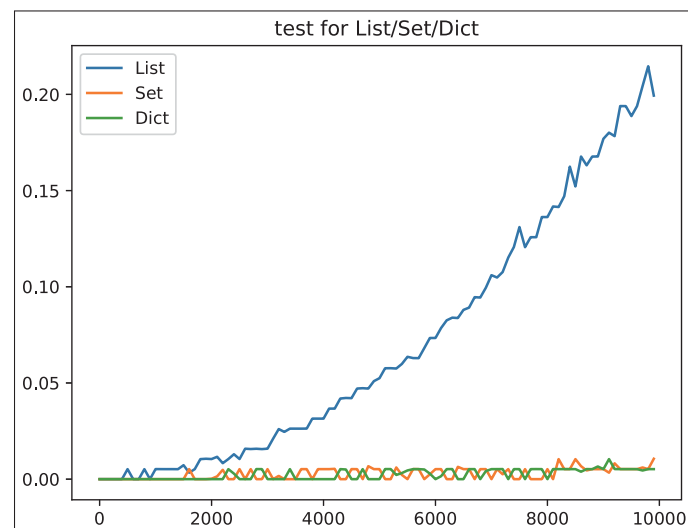


図 74: spdTest02.py の実行結果

横軸はデータの個数、縦軸は実行時間

(評価)

データ個数の増加に対してセットと辞書では検査の実行時間の伸びが小さいのに対して、リストでは概ねデータ個数の 2 乗に比例する形で実行時間が大きくなる。リストに対する in 演算子による要素の探索は、線形探索アルゴリズムと同規模の計算時間となることがわかる。

I.2 ライブラリ使用の有無における計算速度の比較

ここでは、Python 用の代表的な数値計算ライブラリである **NumPy** を使用することで大きな計算速度が得られる例を示す。次に示すプログラム `matmult01.py` は 800×800 の行列同士の積

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,799} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,799} \\ \vdots & \vdots & \ddots & \vdots \\ c_{799,0} & c_{799,1} & \cdots & c_{799,799} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,799} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,799} \\ \vdots & \vdots & \ddots & \vdots \\ a_{799,0} & a_{799,1} & \cdots & a_{799,799} \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,799} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,799} \\ \vdots & \vdots & \ddots & \vdots \\ b_{799,0} & b_{799,1} & \cdots & b_{799,799} \end{pmatrix}$$

を求めるものである。

プログラム：matmult01.py

```
1 # coding: utf-8
2 import time
3
4 W = 800 # 配列のサイズ
5
6 M1 = []; M2 = []; M3 = []
7 # サンプル行列の作成
8 for i in range(W):
9     L1 = []; L2 = []; L3 = []
10    for j in range(W):
11        L1.append(float(i+j))
12        L2.append(float(i-j))
13        L3.append(float(0))
14    M1.append(L1); M2.append(L2); M3.append(L3)
15
16 # 行列の積の計算
17 t1 = time.time()
18 print('start')
19 for i in range(W):
20     for j in range(W):
21         for k in range(W):
22             M3[i][j] += M1[i][k] * M2[k][j]
23 t2 = time.time()
24 print('stop')
25 print('time(sec):', t2-t1)
26
27 # 先頭部分の表示
28 for i in range(5):
29     for j in range(5):
30         print(f'{M3[i][j]:.1f}', ' ', end='')
31     print('')
```

このプログラムではリスト `M1`, `M2`, `M3` で行列を表現している。はじめに `M1`, `M2` に値を設定（6～14 行目）し、それらの積を `M3` に得る（19～22 行目）。

このプログラムを実行した様子を次に示す。

```
start          ←行列の積の計算開始
stop           ←計算終了
time(sec): 132.11081528663635    ←計算にかかった時間
170346800.0, 170027200.0, 169707600.0, 169388000.0, 169068400.0, ←計算結果の
170666400.0, 170346000.0, 170025600.0, 169705200.0, 169384800.0, ←最初の 5 × 5 の
170986000.0, 170664800.0, 170343600.0, 170022400.0, 169701200.0, ←部分
171305600.0, 170983600.0, 170661600.0, 170339600.0, 170017600.0,
171625200.0, 171302400.0, 170979600.0, 170656800.0, 170334000.0,
```

このプログラムを Python 3.11.2, Intel Core i7-1195G7 2.9GHz, 32GB RAM, Windows 11 Pro の環境で 3 回実行して得られた計算時間の平均値は 134.5376 秒であった。

次に、同様の計算を行うプログラムを NumPy を用いて実装した例が次に示す `matmult01_np.py` である。

プログラム：matmult01_np.py

```

1  # coding: utf-8
2  import time
3  import numpy as np
4
5  W = 800 # 配列のサイズ
6
7  M1 = []; M2 = []; M3 = []
8  # サンプル行列の作成
9  for i in range(W):
10     L1 = []; L2 = []; L3 = []
11     for j in range(W):
12         L1.append(float(i+j))
13         L2.append(float(i-j))
14         L3.append(float(0))
15     M1.append(L1); M2.append(L2); M3.append(L3)
16 N1 = np.array(M1); N2 = np.array(M2)
17
18 # 行列の積の計算
19 t1 = time.time()
20 print('start')
21 N3 = np.dot(N1,N2)
22 t2 = time.time()
23 print('stop')
24 print('time(sec):',t2-t1)
25
26 # 先頭部分の表示
27 for i in range(5):
28     for j in range(5):
29         print(f'{N3[i][j]:.1f}', ',end='' )
30     print('')
```

このプログラムの前半部（M1, M2 の作成）は先の matmult01.py と同じであるが、それらを NumPy 独自の配列オブジェクト N1, N2 に変換し、積を N3 に得ている。また、行列の積を求める部分には NumPy の dot 関数を用いている（21 行目）。

（NumPy に関する詳細は公式インターネットサイトをはじめとする他の資料²⁹⁰ を参照のこと）

このプログラムを実行した様子を次に示す。

```

start          ←行列の積の計算開始
stop           ←計算終了
time(sec):  0.015180349349975586    ←計算にかかった時間
170346800.0, 170027200.0, 169707600.0, 169388000.0, 169068400.0,  ←計算結果の
170666400.0, 170346000.0, 170025600.0, 169705200.0, 169384800.0,  ←最初の 5 × 5 の
170986000.0, 170664800.0, 170343600.0, 170022400.0, 169701200.0,  ←部分
171305600.0, 170983600.0, 170661600.0, 170339600.0, 170017600.0,
171625200.0, 171302400.0, 170979600.0, 170656800.0, 170334000.0,
```

このプログラムを先と同じ環境で 3 回実行して得られた計算時間の平均値は 0.0144 秒であった。この値は先のプログラムと比較すると約 9341 倍である。

参考までに、同じ処理を行うプログラムを C 言語で実装（matmult01.c）して実行した例を次に示す。

プログラム：matmult01.c

```

1  #include <stdio.h>
2  #include <time.h>
3
4  #define W 800
5
6  int main() {
7     int i, j, k;
8     static double M1[W][W], M2[W][W], M3[W][W];
9     clock_t t1, t2;
10
11     /* サンプル行列の作成 */
```

²⁹⁰拙書「Python3 ライブラリブック - 各種ライブラリの基本的な使用方法」でも解説しています。

```

12     for( i=0; i<W; i++ ) {
13         for ( j=0; j<W; j++ ) {
14             M1[i][j] = (double)i + (double)j;
15             M2[i][j] = (double)i - (double)j;
16         }
17     }
18
19     /* 行列の積の計算 */
20     printf("start\n");
21     fflush(stdout);
22     t1 = clock();
23     for( i=0; i<W; i++ ) {
24         for ( j=0; j<W; j++ ) {
25             M3[i][j] = 0.0;
26             for ( k=0; k<W; k++ ) {
27                 M3[i][j] += M1[i][k] * M2[k][j];
28             }
29         }
30     }
31     t2 = clock();
32     printf("stop\n");
33     fflush(stdout);
34
35     printf("time(sec): %8.4f\n", (double)(t2-t1) / CLOCKS_PER_SEC );
36
37     /* 先頭部分の表示 */
38     for ( i=0; i<5; i++ ) {
39         for ( j=0; j<5; j++ ) {
40             printf("%.1f, ", M3[i][j]);
41         }
42         printf("\n");
43     }
44
45     return 0;
46 }

```

このプログラムを先と同じ環境の MinGW64 下でコンパイル (gcc 12.2.0, オプション -O3) して実行した様子を次に示す。

```

start          ←行列の積の計算開始
stop           ←計算終了
time(sec):    0.3040    ←計算にかかった時間
170346800.0, 170027200.0, 169707600.0, 169388000.0, 169068400.0, ←計算結果の
170666400.0, 170346000.0, 170025600.0, 169705200.0, 169384800.0, ←最初の 5 × 5 の
170986000.0, 170664800.0, 170343600.0, 170022400.0, 169701200.0, ←部分
171305600.0, 170983600.0, 170661600.0, 170339600.0, 170017600.0,
171625200.0, 171302400.0, 170979600.0, 170656800.0, 170334000.0,

```

3 回実行して得られた計算時間の平均値は 0.304 秒であった。これは最初のプログラム `matmult01.py` と比較すると約 443 倍の実行速度である。ここに挙げた 3 つのプログラムの実行時間などを表 62 にまとめる。

表 62: 実験結果の比較

プログラム	特徴	実行時間 (秒)	速度比 (倍)
matmult01.py	行列をリストで表現	134.5376	(基準) 1
matmult01.np.py	NumPy を用いて計算	0.0144	9341
matmult01.c	C 言語による実装	0.304	443

I.3 os.walk 関数の応用例

os モジュールの walk 関数²⁹¹ を用いると、指定したディレクトリ内の全ての項目（配下のディレクトリ、ファイル）を独自のタプルの形式で列挙することができる。この関数を応用して、指定したディレクトリ内の全ての項目を1つずつ取得するジェネレータを実装する例を次の walkdir.py に示す。

プログラム：walkdir.py

```
1 # coding: utf-8
2 import os
3
4 def walkdir( d ):
5     for dpath, dnames, fnames in os.walk( d ):
6         for fname in fnames:
7             yield os.path.join(dpath, fname)
8         for dname in dnames:
9             yield os.path.join(dpath, dname)
```

walkdir.py に定義された walkdir 関数は、引数に与えられたパス（文字列形式）が示すディレクトリ配下の項目をもたすジェネレータを返す。

例えば、カレントディレクトリに右図のような階層構造（'<...>' はディレクトリ）を持つディレクトリ exdir（p.121 に例示したものと同じ）がある場合の walkdir 関数の実行例を次に示す。

```
<exdir>
|-- <emp>
|-- file1.txt
|-- <subdir>
    |-- file2.txt
    |-- file3.txt
    |-- <subsubdir>
        |-- file4.txt
```

例. walkdir 関数の実行例

```
>>> import walkdir  Enter      ←上記ファイルをモジュールとして読み込む
>>> for x in walkdir.walkdir('./exdir'): Enter    ←ディレクトリ内容の一覧処理
...     print(x)  Enter      ←内容項目の出力
...  Enter      ← for 文の記述の終了
./exdir/file1.txt          ←↓ディレクトリ内の全項目が列挙される
./exdir/emp
./exdir/subdir
./exdir/subdir/file2.txt
./exdir/subdir/file3.txt
./exdir/subdir/subsubdir
./exdir/subdir/subsubdir/file4.txt
```

これは、カレントディレクトリにある exdir ディレクトリに含まれる全ての項目（全てのサブディレクトリ、ファイル）を列挙した例である。

²⁹¹ 「2.7.7.4 ディレクトリ内容の一覧 (2)」(p.121) で解説している。

I.4 pathlib の応用例

pathlib モジュールの機能を使用したサンプルプログラムを示す。指定したディレクトリ配下に存在するパスのうち、指定したパターンに合致するファイル名（あるいはディレクトリ名）のものを探し出す（サブディレクトリ配下も再帰的に探し出す）プログラムの例を示す。

プログラム：pfind.py

```
1  # coding: utf-8
2  # モジュールの読み込み
3  from pathlib import Path
4
5  # 主な処理を行う関数
6  def pfind0( path, ptn ):
7      r = list( path.glob(ptn) )           # まずは当該ディレクトリ内で検索
8      pall = path.glob('*')               # ディレクトリ内の全ての要素を列挙
9      for m in pall:                       # 1つずつ調べながら
10         if m.is_dir():                    # それがディレクトリならば
11             r += pfind0( m, ptn )        # 再帰的にサブフォルダを検索
12     return r                             # リストとして値を返す
13
14 # 入り口となる関数
15 def pfind( p, ptn ):
16     return pfind0( Path(p), ptn )
```

このプログラムはモジュールとして使用することができる。ファイル検索のための関数が pfind として定義されており、

pfind(検索対象のディレクトリ, パターン)

として呼び出す。パターンは glob メソッドに与える形式の文字列であり、glob 独自の正規表現である。pfind 関数は、パターンに合致したパスのリストを戻り値として返す。(次の例参照)

例. Windows 環境の '/Windows' ディレクトリから 'cmd.exe' を探す

```
>>> from pfind import pfind  [Enter]    ←モジュールの読み込み
>>> r = pfind('/Windows','cmd.exe') [Enter] ←検索実行
>>> for m in r: print(m) [Enter]    ←表示処理
... [Enter]                        (記述ここまで)

¥Windows¥System32¥cmd.exe          ←ここから結果表示
¥Windows¥SysWOW64¥cmd.exe
:
(以下省略)
:
```

I.5 浮動小数点数と2進数の間の変換

浮動小数点数から2進数への、あるいはその逆の変換を行うサンプルプログラムを示す。

プログラム：f2bin.py

```
1  # coding: utf-8
2  # float -> 2進数
3  def f2bin(v,n):
4      # 符号取得
5      if v < 0:
6          s = '-'
7          v *= -1
8      else:
9          s = ''
10     vb = bin(round(v*2**n))[2:]      # 左シフトで整数化
11     l = len(vb)
12     if l <= n:                       # 桁の長さ調整
13         vb = '0'*(n-l+1) + vb
14     # 2進数を構成後、右端の'0'を除去する処理
15     r = (s+vb[:-n]+'.'+vb[-n:])[::-1]
16     for i,b in enumerate(r):
17         if b != '0':
18             r = r[i:]
19             break
20     return r[::-1]
21
22 # 2進数 -> float
23 def bin2f(b):
24     # 符号取得
25     if b[0] == '-':
26         s = -1
27         b = b[1:]
28     else:
29         s = 1
30     n = b.index('.')      # 小数点の位置を取得
31     r = int(b[:n],2)      # 小数点より左の値を算出
32     # 小数点以下を算出
33     for i,d in enumerate(b[n+1:]):
34         if d == '1':
35             r += 2**(-1*(i+1))
36     return s*r
```

‘f2bin.py’は2つの関数 f2bin (float 型の値を2進数表記の文字列に変換する)、bin2f (2進数表記の文字列をfloat 型の値に変換する) から成る。

書き方： f2bin(float 型の値, 2進数に変換した際の小数点以下の桁数)
bin2f(小数点付き2進数の文字列)

上記のファイルをカレントディレクトリに配置してモジュールとして扱う。

例. float 型から2進数への変換

```
>>> from f2bin import f2bin, bin2f  Enter    ←関数の読み込み
>>> b = f2bin( 0.1, 52 )  Enter    ← 0.1 を2進数に変換 (小数点以下52ビットで表現)
>>> b  Enter    ←確認
'0.00011001100110011001100110011001100110011001101'    ←変換結果
>>> bin2f(b)  Enter    ← 2進数からfloat型へ変換
0.100000000000000009    ←変換結果
```

注意) 変換に伴う誤差に注意すること。

I.5.1 mpmath を用いた例

mpmath ライブラリを用いると、浮動小数点数を高い精度で 2 進数に変換することができる。基本的には「【参考】浮動小数点数の 2 進数表現」(p.27) で解説した方法 (仮数部と指数部の取り出し) を採用する。

以下に 10 進数表現の 0.1 を 2 進数表現に変換する例を示す。

例. 10 進数の 0.1 を mpf オブジェクトとして用意する

```
>>> from mpmath import mp  ← mpmath の読み込み
>>> mp.dps = 20  ← 計算精度の設定: 10 進数表現で 20 桁
>>> n = mp.mpf('0.1')  ← mpf オブジェクト形式で 10 進数の 0.1 を生成
>>> print( n )  ← 確認
0.1
```

mpf オブジェクトの man, exp プロパティから仮数部と指数部を取り出すことができる。

例. 仮数部と指数部の参照 (先の例の続き)

```
>>> n.man  ← 仮数部の参照
944473296573929042739 ← 10 進数表現
>>> n.exp  ← 指数部の参照
-73 ← 10 進数表現
```

得られた仮数部を bin 関数で 2 進数に変換して表示する例を示す。

例. 2 進数表現 (先の例の続き)

```
>>> print( f'{bin(n.man)}x2**{n.exp}' )  ← 2 進変換と整形表示
0b11001100110011001100110011001100110011001100110011001100110011001100110011001100110011x2**-73
```

このように、10 進数表現の 0.1 が 2 進数の近似表現

$$110011_{(2)} \times 2^{-73}$$

として得られている。

I.6 全ての Unicode 文字の列挙

Unicode はコードポイントの値 0～1114111（16 進数で 0～10FFFF）の範囲で定義されている。ただし本書執筆時点では、この範囲の全てのコードポイントに対しては Unicode 文字は割り当てられていない。

unicodedata モジュールの name 関数を利用して、文字が割り当てられている全ての Unicode 文字を取得するプログラムの例を allunicode.py に示す。

プログラム：allunicode.py

```
1 # coding: utf-8
2 import unicodedata
3
4 all = {}          # 全ての Unicode とその名前の辞書
5 nochr = set()     # 割り当てられていない Unicode
6 for i in range(1114112):
7     c = chr(i)
8     n = unicodedata.name(c, None)
9     if n:
10         all[c] = n
11     else:
12         nochr.add(c)
13
14 # 割り当てられている全ての Unicode 文字のリストを返す関数
15 def allunicode():
16     return sorted(list(all.keys()))
17
18 # テスト実行
19 if __name__ == '__main__':
20     f = open('allunicode_out.txt', 'wb')
21     a = allunicode()
22     for c in a:
23         s = str(ord(c)) + '\t' + c + '\t' + all[c] + '\n'
24         print(s, end='')
25         b = s.encode('utf-8')
26         f.write(b)
27     f.close()
```

このプログラムをスクリプトとして実行すると、文字が割り当てられている全ての Unicode 文字とその属性を表示する。また、モジュールとして読み込んで動作を確認することもできる。

このプログラムをモジュールとして読み込むと、文字が割り当てられている Unicode とその属性が辞書 all に得られる。また、文字が割り当てられていない Unicode のセットが nochr に得られる。

例. 上のプログラムをモジュールとして読み込む例

```
>>> import allunicode Enter    ←プログラムをモジュールとして読み込む
>>> len( allunicode.all ) Enter    ←文字が割り当てられている Unicode の数を調べる
138552    ←文字が割り当てられている Unicode の数（執筆時点の値）
>>> len( allunicode.nochr ) Enter    ←文字が割り当てられていない Unicode の数を調べる
975560    ←文字が割り当てられていない Unicode の数（執筆時点の値）
>>> a = allunicode.allunicode() Enter    ←文字が割り当てられている全 Unicode 文字のリスト
>>> n = a.index('A'); a[n:n+15] Enter    ←部分的に表示
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']    ←対応する文字のリスト
>>> n = a.index('あ'); a[n:n+12] Enter    ←部分的に表示
['あ', 'あ', 'い', 'い', 'う', 'う', 'え', 'え', 'お', 'お', 'か', 'が']    ←対応する文字のリスト
```

文字が割り当てられている Unicode のキャラクタであってもそのためのフォントが導入されていない計算機環境ではその文字を表示（印字）することができないことに留意すること。

J その他

J.1 演算子の優先順位

各種の演算子や括弧などの結合の強さを表 63 に示す。

表 63: 演算子の優先順位： 上の方が結合の優先順位が高い

演 算 子	説 明
(expressions...), [expressions...], { key : value... }, { expressions... }	結合式または括弧式、リスト表示、 辞書表示、集合表示
x[index], x[index:index], x(arguments...), x.attribute	添字指定、スライス操作、呼び出し、 属性参照
await x	Await 式
**	べき乗
+x, -x, ~x	正数、負数、ビット単位 NOT
*, @, /, //, %	乗算、行列乗算、除算、切り捨て除算、剰余
+, -	加算および減算
<<, >>	シフト演算
&	ビット単位 AND
^	ビット単位 XOR
	ビット単位 OR
in, not in, is, is not, <, <=, >, >=, !=, ==	所属や同一性のテストを含む比較 値の比較
not x	ブール演算 NOT
and	ブール演算 AND
or	ブール演算 OR
if – else	条件式
lambda	ラムダ式
:=	代入式

公式インターネットサイト (<https://docs.python.org/>) から引用

J.2 アスキーコード表

表 64: アスキー (ASCII) コードに対応する文字

10進	16進	文字	10進	16進	文字	10進	16進	文字	10進	16進	文字
0	0x00	NUL(null文字)	32	0x20	SPC(空白)	64	0x40	@	96	0x60	`
1	0x01	SOH(ヘッダ開始)	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX(テキスト開始)	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX(テキスト終了)	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT(転送終了)	36	0x24	\$	68	0x44	D	100	0x64	d
5	0x05	ENQ(照会)	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK(受信確認)	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL(警告)	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS(後退)	40	0x28	(72	0x48	H	104	0x68	h
9	0x09	HT(水平タブ)	41	0x29)	73	0x49	I	105	0x69	i
10	0x0a	LF(改行)	42	0x2a	*	74	0x4a	J	106	0x6a	j
11	0x0b	VT(垂直タブ)	43	0x2b	+	75	0x4b	K	107	0x6b	k
12	0x0c	FF(改頁)	44	0x2c	,	76	0x4c	L	108	0x6c	l
13	0x0d	CR(復帰)	45	0x2d	-	77	0x4d	M	109	0x6d	m
14	0x0e	SO(シフトアウト)	46	0x2e	.	78	0x4e	N	110	0x6e	n
15	0x0f	SI(シフトイン)	47	0x2f	/	79	0x4f	O	111	0x6f	o
16	0x10	DLE(データリンクエスケープ)	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1(装置制御 1)	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2(装置制御 2)	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3(装置制御 3)	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4(装置制御 4)	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK(受信失敗)	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN(同期)	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB(転送ブロック終了)	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN(キャンセル)	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM(メディア終了)	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1a	SUB(置換)	58	0x3a	:	90	0x5a	Z	122	0x7a	z
27	0x1b	ESC(エスケープ)	59	0x3b	;	91	0x5b	[123	0x7b	{
28	0x1c	FS(フォーム区切り)	60	0x3c	<	92	0x5c	\	124	0x7c	
29	0x1d	GS(グループ区切り)	61	0x3d	=	93	0x5d]	125	0x7d	}
30	0x1e	RS(レコード区切り)	62	0x3e	>	94	0x5e	^	126	0x7e	~
31	0x1f	US(ユニット区切り)	63	0x3f	?	95	0x5f	_	127	0x7f	DEL(削除)

網掛け部分は、制御文字を含む非印字可能文字である。

92(0x5c) に対応する文字は、日本円記号「¥」として表示される場合もある。

索引

`*`, 13, 85, 139
`**`, 13, 140
`**=`, 13
`*=`, 13
`+`, 13
`+=`, 13, 88
`-`, 13
`-*`, 6
`-help`, 356, 358
`-=`, 13
`-i`, 445
`-h`, 356, 358
`->`, 444
`..`, 58, 110
`...`, 110
`...`, 51
`/`, 13
`//`, 13
`/=`, 13
`∴`, 57, 445
`∴=`, 191
`∴`, 7
`<<`, 54
`<<=`, 55
`=`, 8
`==`, 98
`!=`, 98
`>`, 98
`<`, 98
`<=`, 98
`>=`, 98
`>>`, 54
`>>=`, 55
`[]`, 57
`%`, 13
`%=`, 13
`&=`, 55
`-`, 9, 16, 86
`__abs__`, 163
`__add__`, 163
`__aiter__`, 288
`__and__`, 163
`__anext__`, 288
`__annotations__`, 444
`__bases__`, 363
`__bool__`, 163
`__builtins__`, 137
`__bytes__`, 163
`__ceil__`, 163
`__class__`, 179
`__complex__`, 163
`__contains__`, 167
`__defaults__`, 143
`__del__`, 154, 155
`__delitem__`, 167
`__dict__`, 178
`__divmod__`, 163
`__doc__`, 443
`__enter__`, 341
`__eq__`, 163
`__exit__`, 341
`__file__`, 122, 302
`__float__`, 163
`__floor__`, 163
`__floordiv__`, 163
`__ge__`, 163
`__getattr__`, 165
`__getattribute__`, 166
`__getitem__`, 167
`__getstate__`, 308
`__gt__`, 163
`__iadd__`, 165
`__iand__`, 165
`__ifloordiv__`, 165
`__ilshift__`, 165
`__imatmul__`, 165
`__imod__`, 165
`__imul__`, 165
`__init__`, 154
`__init__.py`, 300, 302
`__int__`, 163
`__invert__`, 163
`__ior__`, 165
`__ipow__`, 165
`__irshift__`, 165
`__isub__`, 165
`__iter__`, 169
`__itruediv__`, 165
`__ixor__`, 165
`__kwdefaults__`, 144

`__le__`, 163
`__len__`, 167
`__lshift__`, 163
`__lt__`, 163
`__main__`, 261, 299
`__matmul__`, 163
`__mod__`, 163
`__mul__`, 163
`__name__`, 52, 187, 261, 299, 362
`__ne__`, 163
`__neg__`, 163
`__new__`, 176
`__next__`, 169
`__or__`, 163
`__pos__`, 163
`__pow__`, 163
`__pycache__`, 300
`__radd__`, 164
`__reduce__`, 308
`__repr__`, 162
`__round__`, 163
`__rshift__`, 163
`__setattr__`, 167
`__setitem__`, 167
`__setstate__`, 308
`__slots__`, 449
`__str__`, 162, 163
`__sub__`, 163
`__subclasses__`, 363
`__truediv__`, 163
`__trunc__`, 163
`__xor__`, 163
`_exit`, 361
`—`, 72
`|`, 54, 79, 251, 447
`|=`, 55
`”`, 72
`$`, 252
`&`, 54
`'''`, 39
`¥`, 7, 38, 39
`^`, 54, 249, 252
`^=`, 55
`@`, 344
`@classmethod`, 157
`@property`, 172
`0b`, 16
`0o`, 16
`0x`, 16
10 進数, 53, 54
16 進数, 16, 54
1 の補数, 54, 55
2 進数, 16, 54
32-bit floating-point の WAV 形式ファイル, 387
8 進数, 16, 54
abs, 17
abspath, 122, 302
accept, 368
acos, 20
acosh, 20
acquire, 258
ActionBar, 226
ActionButton, 226
ActionGroup, 226
ActionPrevious, 226
ActionView, 226
active, 211
active_children, 262
add, 73, 204, 214
add_argument, 356, 357
add_command, 420
add_parser, 360
add_subparsers, 360
add_widget, 204, 206
after, 427
ainput, 283
aioconsole, 283
aiofiles, 288
all, 188
all_tasks, 285
Anaconda, 401
anchor, 421
AnchorLayout, 198, 212
and, 98
ANSI エスケープシーケンス, 39
any, 188
App, 196
append, 58, 323
appendleft, 323
argparse, 356
Argument Parser, 356
argv, 128
Array, 264
as, 339, 433
as_completed, 272

- as_integer_ratio, 29
- ASCII, 47, 466
- asctime, 239
- asin, 20
- asinh, 20
- assert, 337, 338
- AssertionError, 338
- astimezone, 235
- async, 275
- AsyncBufferedIOBase, 290
- AsyncBufferedReader, 290
- asyncio, 275, 383
- asyncio.CancelledError, 286, 375
- asyncio.Future , 281
- AsyncTextIOWrapper, 290
- atan, 20
- atan2, 20
- atanh, 20
- atexit, 361
- auto, 333
- available_timezones, 236
- await, 275, 276
- Awaitable, 275
- b64decode, 316
- b64encode, 316
- Base64, 316
- BaseException, 339
- basename, 122
- BASIC 認証, 376
- Beautiful Soup, 378
- BeautifulSoup, 378
- Bezier, 215
- bin, 54
- bind, 205, 367, 425
- bit_length, 55
- BMP, 48
- bool, 50, 51
- BooleanVar, 412
- BoxLayout, 197, 198, 202
- break, 97
- BrokenPipeError, 375
- buf, 267
- BufferedReader, 109
- BufferedWriter, 109
- build, 196
- Builder, 223
- Button, 197, 202, 210, 409
- Button1, 425
- Button3, 425
- ButtonPress, 425
- ButtonRelease, 425
- bytearray, 317
- bytes, 56, 112, 113
- BytesIO, 319
- calendar, 238
- cancel, 224, 285
- cancel_job, 295
- Canvas, 420
- canvas, 203, 213
- Canvas グラフィックス, 213
- capitalize, 44
- CardTransition, 225
- Carousel, 229
- case, 192
- ceil, 20
- center, 106
- chain, 326
- chardet, 114
- chdir, 120
- CheckBox, 197, 211
- Checkbutton, 411
- choice, 87
- choices, 87
- chr, 47
- class, 154, 325
- clear, 61, 73, 76, 77, 204
- clearcolor, 200
- Clock, 224
- ClockEvent, 224
- close, 111, 267, 284, 353, 368, 373, 383, 387, 394
- cls, 157
- cmath, 21
- coding, 6
- collections, 323
- Color, 204, 214
- color, 208
- combinations, 329
- combine, 231
- Combobox, 413
- CompletedProcess, 380
- complex, 12, 16, 17
- concurrent.futures, 270
- conda, 404
- Config, 201

- config, 420
- Configure, 429
- configure, 418
- conjugate, 17
- connect, 368
- ConnectionRefusedError, 375
- ConnectionResetError, 375
- contents, 379
- Context, 32
- continue, 97
- Control, 425
- cookies, 377
- coords, 423
- copy, 66, 67, 74, 81
- copy2, 352
- copytree, 352
- cos, 20
- cosh, 20
- count, 46, 64, 326
- Counter, 324
- create_arc, 420
- create_image, 422
- create_line, 421
- create_oval, 420
- create_polygon, 421
- create_rectangle, 420
- create_task, 276, 277
- create_text, 421
- CSPRNG, 37
- CSV, 128
- ctime, 233, 238
- currency, 246
- current, 225
- curselection, 416
- cwd, 124
- cycle, 327

- date, 231
- datetime, 231
- day, 234
- Decimal, 31
- decimal, 31
- decode, 112, 113
- dedent, 348
- deep copy, 67
- deepcopy, 67
- def, 137
- DEFAULT_FONT, 210
- del, 10, 60, 76, 153
- delete, 423
- denominator, 29
- deque, 323
- destroy, 432
- destructuring assignment, 85
- detect, 114
- dict, 76
- DictReader, 135
- DictWriter, 131
- difference, 74
- dir, 179
- dirname, 122, 302
- discard, 73
- divmod, 30
- do, 295
- docstring, 443
- Double, 425
- DoubleVar, 412
- down, 212
- drain, 373
- dump, 306
- dumps, 306

- e, 20
- elif, 98
- Ellipse, 204, 215
- Ellipsis, 51
- ellipsis, 51
- else, 91, 97, 98, 188
- Emacs, 6
- encode, 113
- encoding, 110, 377
- endswith, 46
- enter, 291
- Entry, 413
- Enum, 330
- enum, 330
- enum.Flag, 334
- enum.IntFlag, 335
- enumerate, 96, 257
- enumerate オブジェクト, 96
- environ, 120, 350, 406
- EOF, 304
- euc-jp, 6
- eval, 321, 322
- Event, 291
- events, 202

- except, 63, 337
- Exception, 339
- exec, 321
- exists, 121, 124
- exit, 204, 360
- exp, 20, 27, 34
- export_to_png, 217
- extend, 59
- extract, 355
- extractall, 355
- f-string, 104
- factorial, 20
- FadeTransition, 225
- FallOutTransition, 225
- False, 50
- families, 421
- FBO, 217
- FIFO, 61, 323
- files, 437
- fill, 347
- FILO, 61, 323
- filter, 64, 187
- filterwarnings, 440
- finally, 63
- find, 45
- find_all, 379
- finditer, 247
- Flag, 334
- flag, 358
- float, 12, 14, 51
- float.info.epsilon, 15
- float.info.mant_dig, 15
- float.info.max, 15
- float.info.max_exp, 15
- float.info.min, 15
- float.info.min_exp, 15
- FloatLayout, 198
- float の値が整数値かどうかを検査する方法, 53
- float 型に関する各種の情報, 15
- floor, 20
- flush, 117
- font_name, 208
- font_size, 208
- for, 89
- format, 103
- format_exc, 337
- for を使ったデータ構造の生成, 92
- Fraction, 28
- fractions, 28
- Frame, 407
- frames_per_buffer, 397
- from, 300, 301, 433
- from_bytes, 56
- from_float, 28
- frozenset, 75
- Function Annotations, 444
- functools, 190
- Future, 270
- gather, 277
- gauss, 36
- gcd, 20
- geometry, 408
- get, 77, 376, 412
- get_event_loop, 279
- get_extra_info, 374
- get_format_from_width, 394
- get_localzone, 236
- get_localzone_name, 236
- get_name, 286
- get_pixel_color, 218
- get_running_loop, 279
- getattr, 160
- getcontext, 32
- getcwd, 120
- getencoding, 246
- getframerate, 388
- getlocale, 244
- getnchannels, 388
- getnframes, 388
- getpass, 107
- getpid, 350
- getrecursionlimit, 147
- getsampwidth, 388
- getsize, 121
- getsizeof, 441
- getstate, 36
- GET リクエストの送信, 376
- GIL, 259
- glob, 124, 461
- global, 146
- globals, 321, 339
- gmpy2, 27, 311
- GMT, 234
- gmtime, 237

- gnuplot, 382
- Graphics, 204
- grid, 409
- GridLayout, 198
- group, 212, 249, 252
- groupby, 327
- guard, 194
- GUI 構築の形式, 224
- GUI 構築の考え方, 196
- hasattr, 180
- hash, 72
- hashable, 76
- headers, 377
- help, 443
- hex, 54
- home, 124
- horizontal, 202, 211
- hour, 234
- id, 11, 102
- ident, 255
- id (Kv) , 223
- if, 97, 188
- imag, 17, 26
- Image, 197, 212, 214
- immutable, 69
- import, 300, 301, 433
- importlib, 437
- in, 44, 62, 74, 77, 334
- indent, 348
- index, 45, 62
- inf, 22
- input, 107
- insert, 59
- int, 12, 13, 51, 53, 108
- int.from_bytes, 56
- IntEnum, 332
- intersection, 74
- intersection_update, 74
- IntFlag, 335
- IntVar, 412
- io.BytesIO, 319
- io.StringIO, 319
- io モジュール, 107, 108, 118
- IPython, 71
- IP アドレス, 367
- is, 52, 101
- is None, 101
- is not None, 101
- is_active, 397
- is_alive, 256, 261
- is_dir, 124
- is_file, 124
- is_integer, 53
- isabs, 123
- isalnum, 43
- isalpha, 43
- isdecimal, 43
- isdir, 121
- isdisjoint, 74
- isfile, 121
- isfinite, 23
- isinf, 23
- isinstance, 52, 102, 179
- islower, 44
- isnan, 23
- iso2022-jp, 6
- ISO8601, 232, 233, 235
- issubclass, 178, 362
- issubset, 74
- issuperset, 74
- isupper, 44
- items, 81
- iter, 92, 169
- iter_modules, 437
- iterable, 89
- itertools, 326
- j, 16
- join, 42, 123, 255, 260
- Jupyter Notebook, 71
- keycode, 427
- KeyError, 73, 76
- KeyPress, 425
- keys, 79
- keysym, 427
- Kivy Designer, 406
- Kivy 利用時のトラブル, 406
- Kivy 言語, 221
- Label, 196, 197, 208, 409
- LabelBase, 210
- Labelframe, 412
- lambda, 186, 187
- Layout, 197
- lcm, 20, 21

- len, 40, 64, 74, 80
- limit_denominator, 28
- Line, 204, 214
- list, 79, 84
- Listbox, 414, 415
- listdir, 120
- listen, 368
- ljust, 106
- ln, 34
- load, 306
- load_file, 223
- load_next, 229
- load_previous, 229
- loads, 306
- locale, 244
- locale_alias, 245
- locals, 339
- localtime, 237
- Lock, 258
- lock, 258, 264
- log, 20
- log10, 20, 34
- log2, 20
- loop, 230
- lower, 44
- lstrip, 46, 111
- mainloop, 407
- make_archive, 353
- maketrans, 43
- man, 27
- Manager, 265
- map, 183
- mappingproxy, 178
- markup, 208
- master, 432
- match, 192, 251
- match オブジェクト, 247, 248
- math, 19
- matplotlib, 269
- max, 18, 34, 211
- max_workers, 272
- memory-profiler, 441
- Menu, 420
- messagebox, 430
- metadata, 437
- microsecond, 234
- min, 18, 34
- MinGW, 2
- minute, 234
- mkdir, 127
- mktime, 238
- module_finder, 437
- ModuleInfo, 437
- mod 演算, 30
- month, 234
- most_common, 324
- Motion, 425
- mpc, 26
- mpf, 25
- mpmath, 25, 365, 463
- mro, 180
- MSYS, 2
- MULTILINE, 254
- Multiple Assignment, 10
- multiprocessing, 260
- mutable, 69
- mypy, 446
- name, 47, 267, 379, 437
- namedtuple, 325, 349
- NameError, 10
- namelist, 354
- Namespace, 266, 357
- NamespaceProxy, 266
- nan, 22
- new_event_loop, 278
- next, 93, 169
- None, 50
- NoneType, 50
- nonlocal, 150
- normal, 212
- not, 98
- not in, 74
- NoTransition, 225
- now, 231
- numbers, 364
- numbers.Complex, 364
- numbers.Integral, 364
- numbers.Number, 364
- numbers.Rational, 364
- numbers.Real, 364
- numerator, 29
- NumPy, 268
- object, 154, 176, 362
- oct, 54

- on_active, 211
- on_press, 204
- on_release, 204
- on_start, 207
- on_stop, 207
- on_touch_down, 201, 204
- on_touch_move, 201, 204
- on_touch_up, 201, 204
- on_value, 211
- open, 108, 125, 387, 394
- open_connection, 373
- OpenGL, 217, 406
- optional arguments, 356
- or, 98
- ord, 47
- orientation, 202, 211
- os, 120, 406
- os._exit, 361
- os.linesep, 110, 111
- os.sep, 123
- OSError, 375
- pack, 313, 409
- PackagePath, 438
- PageLayout, 198
- parse_args, 356, 357
- pass, 8
- Path, 124
- path, 109, 437
- pathlib, 123
- permutations, 329
- pi, 20
- pickle, 306
- PID, 260, 350
- pid, 260
- PIP, 404
- pip コマンド, 404
- pkg_resources, 436
- pkgutil, 437
- platform, 349
- platform.architecture(), 349
- platform.mac_ver(), 349
- platform.platform(), 349
- platform.processor(), 349
- platform.python_compiler(), 349
- platform.system(), 349
- platform.version(), 349
- pop, 61, 73, 78, 323
- Popen, 381
- popleft, 323
- PortAudio, 394
- positional arguments, 356
- PosixPath, 125
- POST リクエストの送信, 376
- pow, 20, 21
- pprint, 346
- prec, 32
- prettify, 378
- print, 5, 103, 116
- print_help, 358
- ProactorEventLoop, 278
- Process, 260
- process_time, 241
- ProcessPoolExecutor, 270
- product, 328
- ProgressBar, 197, 211
- Progressbar, 418
- project_name, 437
- protocol, 432
- PyAudio, 394
- PYTHON_BASIC_REPL, 4
- Python のインストール, 400
- Python のバージョン情報の取得, 349
- Python ランチャー, 402
- quantize, 33
- quit, 414
- Radiobutton, 412
- raise, 338
- randbelow, 37
- randbits, 37
- randint, 35
- random, 35
- randrange, 35
- range, 90
- raw 文字列, 39
- re, 247
- read, 108, 114, 119, 373, 397
- read_bytes, 126
- read_text, 126
- reader, 132
- readframes, 388, 397
- readline, 108, 110
- readlines, 115
- real, 17, 26
- Rectangle, 204, 214

- recv, 368
- reduce, 190
- register, 210, 361
- RelativeLayout, 198
- release, 258
- remove, 61, 122
- remove_widget, 206
- repeat, 327
- REPL, 3
- replace, 42
- REPL 機能の抑止, 4
- repr, 51, 162
- requests, 376
- requires, 437
- resetwarnings, 440
- resizable, 409
- resource, 209
- resource_add_path, 209
- result, 270
- return, 137
- reverse, 66
- reversed, 66
- rewind, 397
- rfind, 45
- RiseInTransition, 225
- rjust, 106
- rmdir, 122, 127
- root (Kv) , 223
- rotate, 323
- ROUND_05UP, 33
- ROUND_CEILING, 33
- ROUND_DOWN, 33
- ROUND_FLOOR, 33
- ROUND_HALF_DOWN, 33
- ROUND_HALF_EVEN, 33
- ROUND_HALF_UP, 33
- ROUND_UP, 33
- rounding, 32
- rstrip, 46, 111
- run, 196, 275, 278, 291, 380
- run_forever, 284
- run_in_executor, 281
- run_pending, 295
- run_until_complete, 278
- running, 270
- sample, 88
- Scale, 418
- ScatterLayout, 198
- sched, 291
- schedule, 294
- schedule_interval, 224
- schedule_once, 224
- scheduler, 291
- Screen, 197, 199, 224, 225
- ScreenManager, 199, 224
- screenshot, 216
- Scrollbar, 417
- ScrollView, 219
- SDL, 406
- search, 247
- second, 234
- secrets, 37
- seed, 36
- seek, 304
- self, 154–156
- send, 368
- serve_forever, 372
- Session, 377
- set, 72, 84, 412
- set_defaults, 360
- set_event_loop, 285
- set_int_max_str_digits, 14
- setattr, 160
- setdefault, 78
- setframerate, 391
- setlocale, 244
- setnchannels, 391
- setparams, 392
- setrecursionlimit, 147
- setsampwidth, 391
- setsockopt, 367
- setstate, 36
- shallow copy, 67
- shared_memory, 267
- shell, 380
- Shift, 425
- shift-jis, 6
- shift_jis, 110
- shuffle, 87
- shutdown, 270, 372
- shutil, 352
- sign, 26
- sin, 20
- sinh, 20
- size, 200

size_hint, 199, 204
sleep, 242, 276
slice, 83
Slider, 197, 211
SlideTransition, 225
sock_accept, 370
sock_connect, 371
sock_recv, 370
sock_sendall, 370
socket, 367
sort, 65
sorted, 66, 70
source, 212
span, 247, 248
split, 41, 123, 254
splitext, 123
splitlines, 42
spos, 202
sqrt, 20, 34
StackLayout, 198
start, 255, 260
start_server, 372
start_stream, 396
startswith, 46
state, 212
staticmethod, 159
status_code, 377
stdin, 383
stop, 284
stop_stream, 394
StopIteration, 93, 169
str, 51, 113
stream_callback, 396
StreamReader, 373
StreamWriter, 373
strftime, 233, 239
string, 48
StringIO, 319
StringVar, 412
strip, 46
strptime, 232, 239
struct, 313, 391
struct_time, 237
sub, 253
submit, 270
subn, 253
subprocess, 380
subprocess.PIPE, 381

suite, 89
sum, 65
super, 154, 155, 173
swapcase, 44
SwapTransition, 225
Switch, 197, 211
symmetric_difference, 74
SyncManager, 265
sys, 15, 106, 108, 128, 204, 349
sys.exit, 360
sys.modules, 434
sys.path, 434
sys.stderr.buffer, 119
sys.stderr のエンコーディング設定, 118
sys.stdin, 108
sys.stdin.buffer, 119
sys.stdin のエンコーディング設定, 108
sys.stdout, 106
sys.stdout.buffer, 119
sys.stdout のエンコーディング設定, 107
system, 385
SystemExit, 361

TabbedPanel, 228
TabbedPanelItem, 228
tan, 20
tanh, 20
tar, 352, 353, 355
tarfile, 355
Task, 275
Tcl/Tk, 407
TCP/IP, 367
tell, 116
terminate, 262, 394
Text, 417
text, 202, 211, 376
TextInput, 197, 210
TextIOWrapper, 107–109, 119
Texture, 214
texture, 214
texture_size, 212
textwrap, 347
the epoch, 237
The Zen of Python, 436
Thread, 255
thread, 255
thread_time, 241
threading, 255

- ThreadPoolExecutor, 270
- time, 231, 237, 427
- timedelta, 232
- timegm, 238
- timeit, 242
- TimeoutError, 375
- Timer, 243
- timezone, 234
- title, 44, 408
- Tk(), 407
- Tkinter, 407
- to_bytes, 56
- to_integral_value, 34
- ToggleButton, 197, 212
- Toplevel, 428
- trace, 412, 418
- traceback, 337
- transition, 199, 224, 225
- translate, 43
- Triple, 425
- True, 50
- trunc, 20
- try, 63, 337
- tuple, 84
- type, 11, 52, 102, 362, 427
- type 型, 52
- type 型オブジェクト, 362
- typing モジュール, 444, 447
- tzinfo, 234
- tzlocal, 236

- UCS-2, 48
- Unicode, 38
- unicodedata, 47
- Unicode の文字符号化モデル, 47
- unicode もじのけんさく, 251
- uniform, 35
- Union, 447
- union, 74
- UNIX エポック, 237
- UNIX 時間, 237
- unlink, 127, 267
- unpack, 85, 314
- unpack_archive, 353
- update, 74, 78
- upper, 44
- URI, 125
- UTC, 234
- utcnw, 234
- utf-8, 6, 110
- utf-8-sig, 6
- uvloop, 290

- Value, 263
- value, 211
- ValueError, 14, 63
- values, 80
- Variable, 411
- vars, 171, 178
- version, 349, 437
- version_info, 349
- vertical, 202, 211
- Video, 197
- view, 80, 81

- wait, 385
- wait_closed, 373
- walk, 121, 460
- warn, 440
- warning, 440
- warnings, 440
- Wave_read, 387, 397
- Wave_write, 387
- WAV 形式, 387
- wheel, 405
- while, 96
- whl, 405
- Widget, 197, 203, 213
- widget, 427
- Window, 200
- WindowsPath, 125
- winfo_children, 432
- winfo_height, 429
- winfo_screenheight, 429
- winfo_screenwidth, 429
- winfo_width, 429
- winfo_x, 429
- winfo_y, 429
- WipeTransition, 225
- with, 128, 340
- working_set, 436
- wrap, 347
- write, 116, 119, 354, 373, 395
- write_bytes, 126
- write_text, 126
- writeframes, 391
- writeframesraw, 391

writeheader, 131
writelines, 117
writer, 129
writerow, 129
writerows, 129
WWW コンテンツ解析, 376

x_root, 427
XML, 378

y_root, 427
year, 234
yield, 297

ZeroDivisionError, 338
ZIP, 352
zip, 94, 185, 353
ZIP_BZIP2, 353
ZIP_DEFLATED, 353
ZIP_LZMA, 353
ZIP_STORED, 353
ZipFile, 353
zipfile, 353
zip オブジェクトの展開, 95
ZoneInfo, 235

アクションバー, 226
アクセサ, 170, 171
浅いコピー, 67
アスキーコード表, 466
アスキー文字, 47
圧縮処理, 352
アプリケーション終了のハンドリング, 431
アプリケーションの開始と終了, 207
アプリケーションの終了, 204
アラインメント, 104
暗号学的に安全な擬似乱数, 37
アンダースコア, 9
アンパック, 85
暗黙値 (関数の仮引数), 138
アーカイブ, 352
イコール, 8
イコールの連鎖, 10
1 行のみから成るスイート, 90
位置の基準, 421
位置引数, 137, 356
一括判定, 188
イテラブル, 89
イテラブルの繰り返し, 327
イテラブルの連結, 326
イテレータ, 92
イベント, 201
イベントオブジェクト, 427
イベントから得られる座標位置, 218
イベント駆動型, 201
イベントハンドラ, 201
イベントループ, 275, 278
イミュータブル, 69
インスタンス, 154
インスタンスの生成, 154
インスタンス変数, 156, 178
インスタンスメソッド, 156
インデックス, 57
インデックスの範囲, 57
インデント, 7, 89, 90
ウィジェット, 196, 197, 407
ウィジェットツリー, 221
ウィジェットのサイズ設定, 220
ウィジェットの伸縮, 410
ウィジェットの登録と削除, 206
ウィンドウ, 200
ウィンドウに関する情報, 429
ウェブスクレイピング, 376
ウォルラス演算子, 191
エスケープシーケンス, 38
エポック秒, 237
エラー, 63, 338
エラーのハンドリング, 63
エンクロージャ, 151, 152
エンコーディング, 6, 47, 110
エンコーディング情報の取得, 377
エンコーディングの変換処理, 113
演算環境, 32
演算子の優先順位, 465
演算精度の設定, 25
円周率, 20
エントリ, 76, 413
エントリの削除, 76
エントリの順序, 81
応答オブジェクト, 376
大文字／小文字の変換と判定, 44
オブジェクト指向, 58, 154
オブジェクトの繰り返し, 327
オプション引数, 356
折れ線, 214
音声入力デバイス, 397
オーバーライド, 157, 173, 196

改行, 39
改行コード, 110, 111
改行コードの削除, 111
下位サロゲート, 48
拡張クラス, 154
拡張子, 124
加算, 13
仮数部, 14, 15, 26, 27
仮数部の桁数の設定, 25
型, 8
型システム, 362
型昇格規則, 18
型の階層, 362
型の検査, 52
型の変換, 51
型ヒント, 445
カプセル化, 170
「空」値, 100
空の辞書, 101
空の辞書の作成, 76
空のタプル, 101
空文字列, 101
空リスト, 57, 101
仮引数, 137
仮引数の個数, 139
カレントディレクトリ, 109, 120, 124
環境変数, 120, 350
環境変数の参照, 350
関数, 137, 186
関数アノテーション, 444
関数オブジェクト, 186
関数の一斉評価, 183
関数の削除, 153
関数名, 186
外部プログラムとの連携, 380
外部プログラムの標準入力のクローズ, 383
外部プロセスとの同期, 385
ガウス分布, 36
画像, 212
ガベージコレクション, 12, 154
基数の指定, 16
基数の変換, 53
規則 (Kv) , 223
基底クラス, 154
基本多言語面, 48
キュー, 61, 323
共通集合, 74
協定世界時, 234

共役複素数, 17
共有メモリ, 263
局所変数, 146
虚数単位, 16
巨大な整数値, 13
虚部, 17, 26
キー, 76
キーの検査, 77
キーボード入力, 107
キーワード引数, 138, 139, 202
偽, 50
疑似乱数, 36
逆順の要素指定, 83
逆正弦関数, 20
逆正接関数, 20
逆双曲線正弦関数, 20
逆双曲線正接関数, 20
逆双曲線余弦関数, 20
逆余弦関数, 20
行頭や行末でのパターンマッチ, 252
空集合, 101
空白文字の削除, 111
空白文字の除去, 45
クォート文字を含む CSV ファイル, 135
国コード, 244
組合せ, 329
組込み関数, 137
クライアント, 367
クラス, 154
クラス階層, 362
クラスの継承関係, 178
クラスの定義の削除, 181
クラス変数, 157, 178
クラスメソッド, 156, 157
繰り返し, 89, 96
繰り返しの中断とスキップ, 97
繰り返しの表記, 250
クロージャ, 151, 152
偶数丸め, 19
グリニッジ標準時, 234
グループ, 252, 254
グローバル変数, 146, 299
経過時間, 231
警告メッセージ, 440
継承, 154
継承関係, 178
桁数の指定, 104
ゲッター, 171, 172

厳格な位置引数, 142
 厳格なキーワード引数, 142
 言語コード, 244
 現在時刻, 231
 現在のエポック秒の取得, 237
 減算, 13
 構造体, 313
 構造的なパターン, 193
 構造的パターンマッチング, 192
 コマンドサーチパス, 405
 コマンドシェル, 380
 コマンド引数, 128, 356
 コマンド引数の形式, 356
 コマンドライン, 356
 コマンドラインオプション, 356
 コメント, 6
 コルーチン, 275
 コルーチンオブジェクト, 275
 コロン, 57
 コンストラクタ, 154
 コンテナ, 57, 167
 コンテンツの階層構造, 379
 コンテンツの取得, 376
 コンボボックス, 413
 コード体系を調べる方法, 114
 コードポイント, 48
 コールバック, 205
 コールバック関数, 395
 コールバックモード, 395
 合計, 65
 誤差, 18
 再帰代入, 13
 再帰的定義, 147
 最小公倍数, 20
 最小値, 18
 最大公約数, 20
 最大値, 18
 サウンドの再生位置をファイルの先頭に戻す, 397
 サウンド再生の終了の検出, 397
 サウンドの繰り返し再生, 397
 サウンドの再生, 394
 サウンドの入出力, 387
 サウンドの入力, 394, 397
 サブクラス, 363
 サブコマンド, 359
 サブプロセス, 380
 サロゲートペア, 48
 参照カウント, 12
 算術演算, 12
 サンプリング周波数, 387
 サンプリングレート, 387
 サーバ, 367
 シェル変数, 380
 識別値, 102
 四捨五入, 19
 指数関数, 20
 指数表記, 14, 25
 指数部, 14, 15, 26, 27
 自然対数, 20
 シャッフル, 87
 集合論, 71, 74
 集合論の操作, 74
 終了コード, 360, 361
 終了ステータス, 360
 終了の待機, 385
 出力先のリダイレクト, 118
 出力バッファ, 117
 書庫（アーカイブ）の作成, 353
 書庫（アーカイブ）の展開, 353
 書庫ファイル, 352
 書式設定, 103
 処理環境に関する情報の取得, 349
 処理のスケジューリング, 291
 真, 50
 シングルクオート, 39
 シングルトン, 50, 177
 真性乱数, 36
 進捗バー, 211
 真のグローバルの名前空間, 321, 339
 真の乱数, 36
 シンボルの調査, 339
 真理値, 50
 ジェネレータ, 92
 ジェネレータ関数, 297
 ジェネレータ式, 298
 時間差, 232
 時間によるイベント, 224
 時間の計測, 240
 時刻, 231
 時刻情報の分解, 231
 辞書か否かの判定, 82
 辞書型, 76
 辞書の for 文への応用, 91
 辞書の結合, 79
 辞書の更新, 78
 辞書の整列, 82

辞書を他の辞書の内部に展開する, 79
実引数, 137
実部, 17, 26
受信, 368
順次アクセス, 304
順序の反転, 66
順列, 329
上位クラス, 154
上位サロゲート, 48
条件付きカウント, 65
条件付きのパターンマッチング, 194
条件分岐, 97
乗算, 13
剰余, 13, 30
除算, 13
垂直, 202
垂直タブ, 39
垂直配置, 204
スイッチ, 211
水平, 202
水平配置, 204
スイート, 89
数学関数, 19
数値タワー, 18
スクリプト, 3
スクリプトの終了, 360
スクリーン, 199, 224
スクリーンショット, 216
スクリーンマネージャ, 199
スクロールバー, 219, 416, 417
スクロールビュー, 219
スケール, 417, 418
スコープ, 146
スタック, 61, 323
スタティックメソッド, 159, 181
ステレオ音声, 387
ステータスの取得, 377
ストリーム, 394
全てのエントリの削除, 77
スライス, 40, 57, 83
スライスオブジェクト, 83
スライダ, 211
スライドイン, 225
スレッド, 255
スレッド ID, 255
スレッド終了の同期, 255
スレッドの実行, 255
スレッドの生成, 255

スロットベースのクラス, 449
スワイプ, 229
スーパークラス, 154, 363
正確な 10 進演算, 31
正規表現, 247, 249
正規分布, 36
正規乱数, 36
正弦関数, 20
整数, 12
整数除算, 29
整数値の桁数を調べる方法, 14
整数値のビット長を求める方法, 55
正接関数, 20
静的な型付け, 8
静的メソッド, 181
セッション情報, 377
セッタ, 171, 172
セット, 71
セットか否かの判定, 75
セットに変換, 84
セットの生成, 72
セットの複製, 74
遷移の効果, 225
絶対値, 17
絶対パス, 109
絶対パスの文字列, 122
絶対パス／相対パスの判定, 123
ゼロの充填, 104
全要素の削除 (set) , 73
双曲線正弦関数, 20
双曲線正接関数, 20
双曲線余弦関数, 20
送信, 368
相対パス, 109
添字, 40, 83
添字の値の省略, 83
添字の増分, 84
ソケット, 367
ソケットオプション, 367
ソケットの用意, 367
属性の調査, 179
大域変数, 146, 299
対数関数, 20
タイマー, 224, 427
タイムアウト, 368
タイムゾーン, 234, 235
タイムゾーン ID, 235
タイムゾーンの変換, 235

対話モード, 3
タグ, 423
タグの検索, 379
多重継承, 154
多重継承におけるメソッドの優先順位, 180
多重スライス, 168
タスク, 275, 276
タッチ, 201
種, 36
多倍長精度の浮動小数点数, 25
多バイト系文字列の変換, 113
多バイト文字, 48
タブ, 39
タブパネル, 228
タプル, 68
タプルか否かの判定, 71
タプルに変換, 84
第一級オブジェクト, 186
代入式, 191
楕円, 215
ダブルクオート, 39
チェックボタン, 411
チェックボックス, 211
置換, 42, 43, 253
置換処理, 253
チャネル数, 387
中央揃え, 106
中央揃え, 104
長方形, 214
直積集合, 328
通信機能, 367
定数の取り扱い, 331
テキスト, 416, 417
テキスト形式, 109
テキストデータ, 112
テキスト入力, 210
テキストファイル, 112
テキストボックス, 413
テクスチャ, 214
ディスプレイに関する情報, 429
ディレクトリ, 120, 123
ディレクトリ階層の複製, 352
ディレクトリ内容の一覧, 120, 121
ディレクトリの削除, 122
ディレクトリの作成, 127
ディレクトリの部分のみを取り出す, 122
ディレクトリの要素, 124
ディレクトリ名, 124

デカルト積, 328
デコレータ, 157, 344
デストラクタ, 154
デフォルト値, 359
デフォルトフォント, 210
データ構造, 57
データ構造に沿った値の割当て, 85
データ構造のシャッフル, 87
データ構造の生成, 92
データ構造の選択的な部分抽出, 87
データ構造の比較, 99
データ構造の変換, 84
特殊なタプル, 70
トグルボタン, 212
同一性, 102
動的な型付け, 8, 445
ドット, 58
内部関数, 149
内包表記, 92
内包表記によるタプルの生成, 92
長さ, 64
名前空間, 266
名前空間プロキシ, 266
名前の衝突, 231, 434
日本語の変数名, 9
日本語文字列の表示, 208
入出力, 103
任意の精度, 25
ヌルオブジェクト, 50, 101
ネイピア数, 20
排他制御, 257, 258, 264
排他的論理和, 54
配置領域の大きさ, 204
ハイフン, 9
派生クラス, 154
ハッシュ可能, 76
バイエンディアン, 315
バイトオーダー, 314
バイト値, 113
バイト列, 112, 306
バイト列の作成方法, 113
バイナリ形式, 109, 306
バイナリデータ, 112, 313, 391
バイナリファイル, 112
バックスペース, 39
バックスラッシュ, 6
バックログ, 368
パイプ, 380

パス, 109
パスオブジェクト, 124
パスの存在の検査, 124
パスの連結, 125
パスワードで保護された ZIP 書庫, 355
パスワード入力, 107
パターンの検索, 247
パターンマッチ, 251
パッケージ, 1, 299, 300
比較演算子, 98
比較演算子の連鎖, 98
引数, 137
引数に暗黙値を設定する, 138
引数の個数が不定, 139
引数パーサ, 356
非数, 23, 26
ヒストリ, 3
左寄せ, 104, 106
日付, 231
日付, 時刻の書式整形, 233
日付と時刻の生成, 232
非同期処理, 275
非同期のイテラブル, 287
非同期のイテレーション, 287
非同期のイテレータ, 288
非同期のコンテナ, 287
非同期のジェネレータ, 287
標準エラー出力, 117
標準出力, 103
標準入力, 103, 107
ビッグエンディアン, 315
ビット演算, 54
ビットフィールド, 334
ビュー, 80, 81
描画色, 214
ピクセル値の取り出し, 217, 218
ファイナライザ, 154
ファイルオブジェクト, 108
ファイルオブジェクトへの出力, 116
ファイルからの入力, 108
ファイル内でのランダムアクセス, 304
ファイルのオープン, 109
ファイルのクローズ, 111
ファイルのサイズの取得, 121
ファイルの削除, 122
ファイルのシーク, 304
ファイルのパス, 109
ファイルの複製, 352

ファイルへの出力, 116
ファイル名, 124
ファイル, ディレクトリの検査, 121, 124
フェードアウト, 225
フェードイン, 225
フォント指定, 208
フォントのサイズ, 208
フォントの登録, 209
フォントのパス, 208
フォーマット済み文字列リテラル, 104
フォームの送信, 376
フォームフィード, 39
深いコピー, 67
複数行に渡る文字列, 39
複数のウィンドウ, 428
複製, 66
複素数, 12, 16, 21, 26
複素数のノルム, 17
符号, 26
浮動小数点数, 12
浮動小数点数と 2 進数の間の変換, 462
フラグ, 358
フレーム, 388
フレームバッファ, 217
フレームバッファへの描画, 217
部分集合, 74
部分集合の判定, 75
部分文字列, 40
部分リスト, 57
ブロッキングモード, 395
分割代入, 85, 94
分子, 29
文書化文字列, 443
分数, 28
分母, 29
分母の大きさの制限, 28
ブレースホルダ, 103, 105
プログラムの終了, 204, 360, 414
プログラムの実行待ち, 242
プログレスバー, 417, 418
プロセス, 260
プロセス ID, 260, 350
プロセス終了の同期, 260
プロセスの強制終了, 262
プロセスの実行, 260
プロセスの生成, 260
プロパティの調査, 179
プロンプト, 3, 107

プール, 270
平方根, 20
平面, 48
ヘッダー情報, 377
変数, 8
変数の値の交換, 86
変数の解放, 10
変数名に関する注意, 9
乗乗, 13
ベル, 39
補助平面, 48
ホームディレクトリ, 120, 124, 350
ホームドライブ, 350
ボタン, 210
ポート, 367
マルチスライス, 168
マルチスレッド, 255
マルチタッチの無効化, 200
マルチバイト文字, 48
マルチプロセッシングマネージャ, 265
マルチプロセス, 260
丸め, 18
マークアップ, 208
右寄せ, 104, 106
ミドルエンディアン, 315
ミュータブル, 69, 76
無限大, 22, 26
無限のカウンタ, 326
無作為抽出, 87
メイン部, 301
メソッド, 154
メソッドの定義, 156
メソッドの適用, 58
メッセージボックス, 430
メニュー, 226, 419
メニュー項目, 419
メニューバー, 226, 419
メンバ, 157
メンバシップ検査, 62
文字コード, 47
文字コード体系, 6, 47
文字コードを検索パターンに使用方法, 250
文字の種類別, 47
文字の置換, 43
文字化け, 112
文字符号化スキーム, 48
文字符号化方式, 48
モジュロ演算, 30
モジュール, 1, 299
モジュールの作成, 299
モジュールの実行, 301
文字列, 38
文字列の含有検査, 44
文字列の繰り返し, 41
文字列の検索, 45, 247, 248
文字列のシャッフル, 87
文字列の置換, 42
文字列の分解, 41, 254
文字列の連結, 41
文字列リテラル, 38
戻り値, 137
モノラル音声, 387
ユーザ定義関数, 137
要素でない, 74
要素の書き換え, 58
要素の個数のカウント, 64
要素の合計, 65
要素の削除, 60, 61
要素の削除 (set), 73
要素の整列, 65, 66
要素の挿入, 59
要素の探索, 62
要素の追加, 58
要素の追加 (set), 73
余弦関数, 20
読み込まれているライブラリの調査, 434
予約語, 9
ライブラリ, 1
ライブラリ管理, 404
ライブラリのパス, 434
ラジオボタン, 411
ラベル, 208
ラベルフレーム, 412
乱数, 35
ランダムアクセス, 304
ランダムサンプリング, 87
リサイズの禁止, 220, 409
リスト, 57
リストか否かの判定, 68
リストに対する検査, 62
リストに変換, 84
リストの拡張, 59
リストの繰り返し, 59
リストの内部に他のリストを展開する方法, 68
リストの長さ, 64
リストの編集, 58

リストの要素の連結, 42
リストの要素へのアクセス, 57
リストの連結, 59
リストボックス, 414, 415
リソース, 209
リテラル, 9, 193
リトルエンディアン, 315
量子化ビット数, 387
両端の文字の除去, 46
累算加算演算子, 88
累算代入, 13, 41, 88, 165
累算代入演算子, 97
ルートディレクトリ, 109
ルートフォルダ, 109
レイアウト, 197
例外, 338
例外オブジェクト, 338
例外処理, 63, 337
列挙型, 330
連結, 59
連続要素のグループ化, 327
ロケール, 244
ロケールのカテゴリ, 244
ロックオブジェクト, 258
論理演算子, 98
論理積, 54
論理和, 54
ローカルタイム, 234
ローカルタイムゾーン, 236
ローカル変数, 146
ワイプ, 225
和集合, 74

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．

本書の内容に関して多くのご指摘とご助言をくださった中西康二様，ここにお礼申し上げます．

本書をご精読くださり，細かな点に至るまでチェックをしてくださいました鈴鹿医療科学大学の梶山純先生，大変感謝しております．ここにお礼申し上げます．

大学，大学院，専門学校での演習授業やゼミにおいて寄せられた相談や質問に端を発する内容が本書の充実に貢献しています．京都情報大学院大学，京都コンピュータ学院，武庫川女子大学の学生諸君に感謝します．また，京都情報大学院大学での教育活動におきまして，今井正治教授（大阪大学名誉教授）との共同活動から新たな知見が得られ，本書の内容の増補に役立ちました．ここにお礼申し上げます．

その他，ご助力くださった多くの方々にもお礼申し上げます．今後とも協力いただけましたら幸いです．また，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．

「Python3 入門」

ー Kivy による GUI アプリケーション開発, サウンド入出力, ウェブスクレイピング

IDEJ 出版 ISBN978-4-9910291-3-4 C3004

著者：中村勝則

発行：2019 年 3 月 14 日	第 1 版
2020 年 3 月 14 日	第 1.3.1 版
2021 年 3 月 14 日	第 2 版
2023 年 3 月 14 日	第 3 版
2023 年 9 月 14 日	第 3.1 版
2024 年 3 月 14 日	第 3.2 版
2025 年 3 月 14 日	第 3.3 版
2025 年 9 月 16 日	第 3.3.16 版

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています.

→ <https://qiita.com/KatsunoriNakamura/items/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させていただいて結構です. (内容を改変せずをお願いします) 内容に関して不備な点がありましたら，是非ご連絡ください. ご意見，ご要望も受け付けています.

● 連絡先

nkatsu2012@gmail.com

中村勝則