

Title	プログラミング演習 Python 2021( Version 2021/10/08 )
Author(s)	喜多, 一; 森村, 吉貴; 岡本, 雅子
Citation	(2021): 1-239
Issue Date	2021-10-08
URL	<a href="http://hdl.handle.net/2433/265459">http://hdl.handle.net/2433/265459</a>
Right	本書はCC-BY-NC-ND(Creative Commons Attribution-NonCommercial-NoDerivatives)ライセンスによって許諾されています。ライセンスの内容を知りたい方は <a href="https://creativecommons.org/licenses/by-nc-nd/4.0/deed.ja">https://creativecommons.org/licenses/by-nc-nd/4.0/deed.ja</a> でご確認ください。
Type	Learning Material
Textversion	publisher

プログラミング演習 Python 2021

Python の予約語、薄い字のものは本書では扱いません

京都大学 国際高等教育院 喜多 一

京都大学 情報環境機構 森村吉貴

京都大学 高等教育研究開発推進センター 岡本雅子

Version 2021/10/08

# 目次

---

目次 .....	2
図目次 .....	7
表目次 .....	10
プログラム目次 .....	11
演習目次 .....	14
<b>0. まえがき .....</b>	<b>17</b>
0.1    目的と到達目標 .....	17
0.2    屋上屋を重ねる理由 .....	17
0.3    文科系がんばれ .....	18
0.4    本書の構成について .....	18
0.5    本書での表記 .....	18
0.6    コピペに注意 .....	19
0.7    2020 年度版からの変更 .....	19
謝辞 .....	20
<b>1. コンピュータとプログラミング .....</b>	<b>21</b>
1.1    この章の目的 .....	21
1.2    コンピュータとプログラム .....	21
1.3    コンピュータの仕組み .....	24
1.4    プログラミング言語 .....	26
1.5    プログラミング言語 Python .....	28
1.6    さまざまな応用 .....	29
1.7    プログラミングの学び方 .....	30
1.8    プログラムを構成する基礎的な概念 .....	36
1.9    プログラムの「どこ」を作るか .....	36
参考文献 .....	37
<b>2. Python の実行環境と使い方 .....</b>	<b>38</b>
2.1    本章の学習の目標 .....	38
2.2    学習環境の想定 .....	38
2.3    準備 .....	39
2.4    IDLE の起動 .....	39
2.5    Python シェル .....	40
2.6    スクリプトの作成と実行 .....	43
2.7    Anaconda Prompt での作業フォルダの設定 .....	45
2.8    IDLE のキー操作など .....	47
2.9    拡張子の表示 .....	47
2.10   Python コマンドの実行 .....	48

2.11	Python を学ぶ環境づくり .....	50
2.12	Mac ユーザへ .....	51
	参考文献 .....	54
<b>3.</b>	<b>変数と演算、代入 .....</b>	<b>55</b>
3.1	本章の学習の目標 .....	55
3.2	プログラムの実行の流れと情報の流れ .....	55
3.3	変数の名前 .....	56
3.4	変数への代入と値の評価 .....	58
3.5	代入演算子 .....	59
3.6	Python で使えるデータ型 .....	60
3.7	Python の変数のより正しい理解 .....	62
3.8	例題：平方根を求める .....	63
3.9	割り算に注意 .....	66
3.10	読み易い式の表記 .....	66
3.11	複数の変数への代入 .....	67
	参考文献 .....	67
<b>4.</b>	<b>リスト .....</b>	<b>69</b>
4.1	本章の学習の目標 .....	69
4.2	Python Shell を用いた学習 .....	69
4.3	リストとは .....	69
4.4	リストの生成 .....	70
4.5	メソッド .....	71
4.6	リストの要素へのアクセス .....	72
4.7	負の添え字とスライス .....	73
4.8	リストへの追加、結合 .....	73
4.9	リストの代入と複製 .....	74
4.10	イミュータブルとミュータブル .....	76
4.11	浅いコピー、深いコピー .....	78
4.12	リストを可視化する .....	79
4.13	計算の過程をリストに残す .....	79
4.14	タプルと辞書 .....	80
<b>5.</b>	<b>制御構造 .....</b>	<b>84</b>
5.1	本章の学習の目標 .....	84
5.2	for 文による繰り返し処理 .....	84
5.3	while 文による繰り返し .....	92
5.4	if 文による分岐 .....	94
5.5	端末からの入力 .....	97
5.6	エラーへの対処 .....	98
5.7	Python での数学関数 .....	99
5.8	数値データ・文字列の変換、文字列の結合 .....	100
5.9	数値を表示する際のフォーマット指定 .....	101
5.10	力試し .....	102
<b>6.</b>	<b>京都の交差点を作る .....</b>	<b>103</b>
6.1	本章の学習の目的 .....	103
6.2	京都の交差点を作る .....	103

6.3	リストのリストとその走査.....	105
<b>7.</b>	<b>関数を使った処理のカプセル化 .....</b>	<b>109</b>
7.1	本章の学習の目標 .....	109
7.2	絶対値関数を作つてみる .....	109
7.3	関数定義の書式 .....	110
7.4	仮引数と実引数 .....	111
7.5	返り値.....	111
7.6	前章の例題から .....	112
7.7	関数 <code>square_root()</code> を実装する .....	113
7.8	関数内の変数の扱い.....	114
7.9	関数の利用パターン .....	115
7.10	関数の呼び出しと関数オブジェクトの引き渡し .....	116
7.11	デフォルト引数値とキーワード引数 .....	117
<b>8.</b>	<b>Turtle で遊ぶ .....</b>	<b>119</b>
8.1	本章の学習の目標 .....	119
8.2	モジュール .....	119
8.3	Turtle –由緒正しき亀さん .....	120
8.4	Python の Turtle モジュール.....	121
8.5	使ってみよう .....	121
8.6	Turtle モジュールの主な関数.....	123
8.7	複数のタートルを動かす .....	124
8.8	作品作りのためのヒント .....	125
8.9	Turtle Demo .....	128
8.10	課題 Turtle の作品制作.....	128
8.11	スクリーンショットの撮り方 .....	128
	参考文献 .....	129
<b>9.</b>	<b>Tkinter で作る GUI アプリケーション(1).....</b>	<b>133</b>
9.1	本章の学習の目標 .....	133
9.2	GUI とイベント駆動型プログラミング .....	133
9.3	モデルとユーザーインターフェイスの分離 .....	134
9.4	<code>tkinter</code> .....	135
9.5	<code>tkinter</code> の例題( <code>tkdemo-2term.py</code> ).....	136
9.6	<code>tkinter</code> を用いたプログラムの基本構成.....	139
9.7	<code>grid</code> によるレイアウト .....	140
9.8	<code>lambda</code> ( $\lambda$ ) 表現を使った Call Back 関数の記述.....	141
9.9	ウィジェットの体裁の調整.....	143
9.10	<code>tkinter</code> の終わり方 .....	146
9.11	<code>Frame</code> クラスを拡張する方式での実装法 .....	147
	参考文献 .....	149
<b>10.</b>	<b>Tkinter で作る GUI アプリケーション(2).....</b>	<b>150</b>
10.1	本章の学習の目標 .....	150
10.2	自律的に動作するプログラムと GUI との衝突 .....	150
10.3	<code>tkinter</code> を用いたアナログ時計プログラム .....	151
10.4	変数を介した動作の協調 .....	158

<b>11. クラス .....</b>	<b>160</b>
11.1 本章の学習の目標 .....	160
11.2 オブジェクト指向プログラミング .....	160
11.3 Python でのクラスの書き方、使い方 .....	161
11.4 クラスの変数とアクセスの制限.....	163
11.5 繙承 .....	166
11.6 インスタンスを起点にクラスを設計する .....	166
<b>12. ファイル入出力 .....</b>	<b>168</b>
12.1 本章の学習の目標 .....	168
12.2 データを永続的に利用するには.....	168
12.3 ファイルについて .....	168
12.4 まずは動かしてみよう .....	171
12.5 Python でのファイルの読み書き .....	172
12.6 例題1 波の近似 .....	174
12.7 例題2 テキストエディタ .....	179
<b>13. 三目並べで学ぶプログラム開発.....</b>	<b>183</b>
13.1 本章の学習の目標 .....	183
13.2 プログラムを開発するということ .....	183
13.3 設計手順—コンピュータを使う前にすること .....	183
13.4 三目並べを例にしたプログラムの設計.....	184
13.5 プログラムの実装 .....	190
13.6 力試し .....	204
13.7 プログラムの開発に関連するいくつかの話題.....	205
<b>14. Python の学術利用.....</b>	<b>207</b>
14.1 本章の学習の目標 .....	207
14.2 import 時の別名 .....	207
14.3 NumPy.....	208
14.4 Matplotlib.....	211
14.5 pandas .....	219
14.6 課題 .....	224
参考文献 .....	226
<b>15. 振り返りとこれから .....</b>	<b>227</b>
15.1 本章の学習の目標 .....	227
15.2 振り返り .....	227
15.3 守破離.....	227
15.4 Python の利用環境 .....	227
15.5 モジュール等の追加.....	228
15.6 本書で紹介しなかった話題 .....	228
15.7 感謝と恩返し—学んだことをどう活かすのか.....	228
<b>16. IDLE Python 便利帳 .....</b>	<b>230</b>
16.1 Python 便利メモ .....	230
16.2 ファイル名に注意 .....	230
16.3 IDLE メモ—Python シェルのキー操作 .....	230

16.4 IDLE メモーエディタ .....	231
<b>17. IDLE/Python でのエラーメッセージの読み方 ..... 232</b>	
17.1 IDLE エディタが表示するエラー.....	232
17.2 実行時に Python Shell で表示されるエラー.....	236

# 図目次

---

図 1-1 ジャカード織機と解析エンジン .....	22
図 1-2 論理素子の進歩 .....	23
図 1-3 マイクロプロセッサ上のトランジスタ数 .....	24
図 1-4 コンピュータ（ハードウェア）の構成 .....	25
図 1-5 プログラミング言語と処理系 .....	26
図 1-6 プログラミング言語処理系の構成方式 .....	28
図 1-7 JIS キーボードの配置 .....	34
図 1-8 フレームワークとライブラリ .....	37
図 2-1 プログラム保存用フォルダの作成 .....	39
図 2-2 Anaconda Prompt からの idle の起動 .....	40
図 2-3 IDLE の Python シェル .....	41
図 2-4 Python Shell での操作 .....	42
図 2-5 IDLE Editor, Python シェルとの違いに注意 .....	44
図 2-6 IDLE での Shell とエディタの連携 .....	45
図 2-7 対話モードでの Python の起動 .....	49
図 2-8 スクリプトを指定した Python の実行 .....	49
図 2-9 -i オプションでスクリプト実行後に対話モードを継続 .....	49
図 2-10 IDLE からオンラインマニュアルの起動 .....	50
図 2-11 Python のオンラインマニュアル（右は日本語を指定した場合） .....	50
図 2-12 Mac での逆スラッシュの入力 .....	53
図 3-1 変数についてのイメージ、代入と評価 .....	59
図 3-2 Python では変数の内容の型は自由です .....	62
図 3-3 Python の変数はデータ（オブジェクト）の所在情報を持つ .....	63
図 3-4 平方根の近似計算の直感的説明 .....	64
図 4-1 リストの代入 .....	76
図 4-2 リストのコピー作成と代入 .....	76

図 4-3 Python Tutor でリストの動作を確認する .....	79
図 7-1 実引数と仮引数 .....	111
図 7-2 グローバル変数とローカル変数 .....	115
図 7-3 関数の利用パターン .....	116
図 7-4 関数呼び出しと「副作用」 .....	116
図 8-1 random_turtle.py の実行結果 .....	127
図 8-2 turtle-tree.py の実行結果 .....	127
図 8-3 Turtle Demo の実行方法 .....	128
図 8-4 スクリーンショットを撮るための Windows でのキー操作 .....	129
図 9-1 イベント駆動型プログラミングの枠組み .....	134
図 9-2 Model-View-Control アーキテクチャ .....	135
図 9-3 Tkinter のシステム構成 .....	135
図 9-4 tkinter でのオブジェクトの関係 .....	140
図 9-5 grid によるレイアウト .....	140
図 10-1 tkinter での after の利用 .....	150
図 10-2 作成するアナログ時計 .....	151
図 10-3 時計の針先位置の計算 .....	152
図 10-4 変数を介した動作の協調 .....	159
図 11-1 クラス変数とインスタンス変数 .....	166
図 12-1 のこぎり波の三角関数の和での近似 .....	174
図 12-2 のこぎり波の三角関数の和での近似（原点で傾きが正の場合） .....	175
図 12-3 「各時刻のデータのリスト」のリストとして扱う .....	179
図 12-4 「各系列のリスト」のリストとして扱う .....	179
図 13-1 三目並べ、棋譜の例 .....	185
図 13-2 着目する手番の勝ち判定 .....	188
図 13-3 勝敗判定 .....	189
図 13-4 ソースコードの全体構成 .....	191
図 13-5 ソフトウェア開発の V モデル .....	206
図 14-1 NumPy, Matplotlib, pandas の関連 .....	207

図 14-2 Matplotlib の使用例 .....	213
図 14-3 散布図の描画 .....	215
図 14-4 ヒストグラムの描画 .....	216
図 14-5 複数のグラフの描画 .....	218
図 14-6 pandas でのグラフ作成 .....	223
図 14-7 べき乗のグラフとのこぎり波の三角関数の和での近似 .....	225

## 表目次

---

表 1-1 プログラミングで用いる記号と読み .....	34
表 2-1 Python の算術演算 .....	42
表 3-1 Python の代入演算子 .....	60
表 3-2 Python で使えるデータ型 .....	61
表 4-1 リスト, タプル, 辞書, 表記のまとめ .....	82
表 5-1 Python の比較演算子 .....	95
表 5-2 数値と文字列の相互変換 .....	100
表 5-3 文字列の結合と繰り返し .....	101
表 4 スクリーンショットの撮り方 .....	129
表 9-1 tkinter での色指定 .....	144

## プログラム目次

---

プログラム 2-1 (p2-1.py) .....	44
プログラム 3-1 平方根を求めるプログラム（その 1, p3-1.py） .....	65
プログラム 4-1 リストを用いた計算過程の保存 (p4-1.py) .....	79
プログラム 5-1 平方根を求めるプログラム（その 2, p5-1.py） .....	85
プログラム 5-2 平方根を求めるプログラム（その 2, p5-2.py） .....	86
プログラム 5-3 continue と break (p5-3.py) .....	87
プログラム 5-4 合計の計算 (p5-4) .....	89
プログラム 5-5 for 文の入れ子 (p5-5.py) .....	89
プログラム 5-6 平方根を計算するプログラム（その 3, p5-6.py） .....	92
プログラム 5-7 平方根を求めるプログラム（無限ループ型, p5-7.py） .....	93
プログラム 5-8 複合的な条件を用いた分岐 (p5-8.py) .....	97
プログラム 5-9 if 文を入れ子にした分岐 (p5-9.py) .....	97
プログラム 5-10 入力を得て検査するプログラム(inputcheck.py) .....	98
プログラム 6-1 (p6-1.py) .....	103
プログラム 6-2 (p6-2.py) .....	106
プログラム 6-3 (p6-3.py) .....	107
プログラム 7-1 絶対値関数の実装例 (p7-1.py) .....	109
プログラム 7-2 絶対値関数の実装例（その 2, 可能なところで return する, p7-2.py） .....	110
プログラム 7-3 関数 square_root() の実装, p7-3 .....	113
プログラム 8-1 turtle を使う例（turtle.py という名前で保存してはいけない, p8-1.py） .....	121
プログラム 8-2 n 角形を描くプログラム（未完成, p8-2.py） .....	122
プログラム 8-3 複数のタートルを動かす (p8-3.py) .....	124
プログラム 8-4 タートルグラフィクスでのマウスクリックへの応答 (p8-4.py) ...	125
プログラム 8-5 random_turtle.py .....	130
プログラム 8-6 detour.py .....	131

プログラム 8-7 turtle-tree.py .....	132
プログラム 9-1 加算のみの電卓(tkdemo-2term.py).....	136
プログラム 9-2 lambda 式を使った引数付きコールバック関数の設定 (tkdemo-2term_lambda.py).....	141
プログラム 9-3 Frame クラスを拡張する tkinter の実装法 (tkdemo-2term_frame_extention.py) .....	147
プログラム 10-1 tkinter でのアナログ時計 (ボタンなし, tkdemo_simple_clock.py)	152
プログラム 10-2 tkinter でのアナログ時計 (ボタンあり, tkdemo_clock_with_button.py) .....	154
プログラム 11-1 CUI 型電卓プログラム (p11-1.py) .....	161
プログラム 11-2 クラス変数とインスタンス変数 (p11-2.py) .....	164
プログラム 12-1 ファイル入出力の例題 (p12-1.py).....	171
プログラム 12-2 のこぎり波の三角関数の和での近似 (p12-2.py).....	175
プログラム 12-3 tkinter を用いた簡単なテキストエディタ(p12-3.py) .....	180
プログラム 13-1 三目並べのプログラム, 実装例 (その 1 グローバル変数) ....	192
プログラム 13-2 三目並べのプログラム, 実装例 (その 2 手番関連の関数) ....	193
プログラム 13-3 三目並べのプログラム, 実装例 (その 3 盤面関連の関数その 1)	194
プログラム 13-4 三目並べのプログラム, 実装例 (その 4 盤面関連の関数その 2)	196
プログラム 13-5 三目並べのプログラム, 実装例 (その 5 盤面のテスト関数 1)	198
プログラム 13-6 三目並べのプログラム, 実装例 (その 6 盤面のテスト関数 2)	200
プログラム 13-7 三目並べのプログラム, 実装例 (その 7 棋譜関連の関数) ....	202
プログラム 13-8 三目並べのプログラム, 実装例 (その 8 プレイ関数とメインプログラム) .....	203
プログラム 14-1 use_matplotlib.py .....	214
プログラム 14-2 use_matplotlib_scatter.py .....	215
プログラム 14-3 use_matplotlib_hist.py.....	216
プログラム 14-4 use_matplotlib_subplot.py .....	218

プログラム 14-5 use_read_csv.py.....	222
プログラム 14-6 use_DaDaFrame_plot.py.....	223
プログラム 14-7 Numpy と Matplotlib で 1~4 乗のグラフを描く (use_matplotlib_power_function.py) .....	225
プログラム 17-1 missing_colon_error.py.....	232
プログラム 17-2 missing_parentheses_error.py .....	233
プログラム 17-3 insufficient_indentation_error.py .....	234
プログラム 17-4 excess_indentation_error.py.....	235
プログラム 17-5 referencing_undefined_variable_error.py .....	236
プログラム 17-6 wrong_argument_type_error.py.....	237
プログラム 17-7 incorrect_indatation_in_class_error.py.....	238
プログラム 17-8 incorrect_optional_argument_error.py.....	239

## 演習目次

---

演習 1-1 この授業の受講動機 .....	21
演習 1-2 プログラミングで使う記号 .....	36
演習 2-1 算術演算の確認 .....	42
演習 2-2 IDLE の Python Shell と Editor の違いの確認 .....	44
演習 2-3 p2-1.py の実行後の確認 .....	45
演習 2-4 あなた自身が Python を学ぶ環境を用意し、報告してください. ....	50
演習 2-5 本日の演習内容をご自身の学習環境で再確認してください. ....	50
演習 3-1 プログラムと楽譜 .....	55
演習 3-2 さまざまな変数名を利用する練習. ....	58
演習 3-3 変数の動作の説明 .....	59
演習 3-4 データの型の確認 .....	61
演習 3-5 平方根を求めるプログラムの作成と実行 .....	65
演習 3-6 エラーを体験する(1). ....	66
演習 3-7 他の数値の平方根を求める. ....	66
演習 4-1 平方根の計算過程のリストへの保存 .....	80
演習 5-1 平方根を求めるプログラムへの for 文の適用 .....	85
演習 5-2 ブロックの確認 .....	86
演習 5-3 イタズラ .....	86
演習 5-4 エラーを体験する(2). ....	87
演習 5-5 エラーを体験する(3). ....	87
演習 5-6 continue と break の説明 .....	88
演習 5-7 range() 関数 .....	89
演習 5-8 合計の計算 .....	89
演習 5-9 for 文の入れ子 .....	90
演習 5-10 リストの要素の平均値を求める .....	91
演習 5-11 要素の参照方法の変更 .....	92
演習 5-12 プログラム 5-6 の作成と実行 .....	93

演習 5-13 エラーを体験する(4).	97
演習 5-14 平方根を求める値の端末からの入力	98
演習 5-15 エラー処理の確認	99
演習 5-16 math モジュールの使用	100
演習 5-17 力試し	102
演習 6-1 京都の交差点の表の作成	106
演習 6-2 リストのリストの出力	108
演習 7-1 絶対値関数を作る	110
演習 7-2 エラーを体験する(5).	110
演習 7-3 平方根を求めるプログラムの関数化	114
演習 7-4 平方根を求めるプログラムの関数化, その 2	114
演習 8-1 正 n 角形を描く	122
演習 8-2 星形の描画	122
演習 8-3 正 7 角形, 正 9 角形とそこでの星形	123
演習 9-1 Tkinter での加算電卓の作成	144
演習 9-2 ウィジェットの体裁の調整	145
演習 9-3 電卓の四則演算への拡張（力試し）	145
演習 9-4 ウィジェットのリストでの管理（力試し）	146
演習 9-5 実際の電卓との差異	146
演習 10-1 アナログ時計で使用するメソッドなどの確認	157
演習 10-2 アナログ時計の改造	158
演習 10-3 表示の改善	158
演習 11-1 Dentaku クラスの拡張	163
演習 11-2 複数のオブジェクトの生成と利用	163
演習 11-3 tkinter で作成した電卓プログラムでの Dentaku クラス利用	163
演習 12-1 矩形波（方形波）の近似	178
演習 12-2 例題 1 のリストを使った実装	178
演習 13-1 複雑な条件判定の書き方	190
演習 13-2 棋譜の採取	204

演習 14-1 Matplotlib でのこぎり波のフーリエ近似の描画 ..... 224

# 0. まえがき

---

本書は京都大学の全学共通科目として実施されるプログラミング演習（Python）の教科書として作成されたものです。

## 0.1 目的と到達目標

この授業の目的・到達目標は以下のように定めています。

### 0.1.1 目的

プログラミング言語 Python は初学者にも学びやすい言語である一方で、さまざまな応用も可能です。近年では学術研究にも利用が広がっています。この授業ではプログラミングの初学者を対象に Python を用いたプログラミングを演習方式で学んでいただきます。

### 0.1.2 到達目標

- Python によるプログラムの実行についての基本操作ができるようになる。
- Python プログラムを構成する基本的要素の機能と書式について説明し、例題を用いて実行例を構成できるようになる。
- Python を用いて簡単なプログラムを自ら設計、実装、テストできるようになる。

## 0.2 屋上屋を重ねる理由

本書は 2018～2020 年度の授業実践に基づいて執筆しました。Python についてはすでに多くの入門書が刊行されているのですが、以下のような理由から、屋上屋を重ねるように教科書を作成しています。

- この授業は Python というプログラミング言語を紹介するのではなく、Python というプログラミング言語で実際にプログラムを書く（書けるようになる）ことを目的にしています。多くの解説書がプログラミング言語の紹介に終始しがちです。

- 初学者にとってプログラミング言語の学習はさまざまな躓きを乗り越えることです。初学者にとっては深刻な躓きでも、ある程度プログラミングの経験を積むと躓いたことさえ忘れてしまいます。この教科書は授業実践を通じて初学者が躓く点やそれへの手助けを意識して記述しています。
- 上記とも関連しますが、プログラミング言語の学習は実際に手を動かしてプログラムを書き、実行することで成り立ちます。本書では実際の演習への指示を示しています。

本書は学習の方向付けとして Python を用いたプログラミングの基本を解説していますが、プログラミング言語の仕様を網羅的に紹介するものではありません。別途、Python についての解説書などを用意して受講されることをお勧めします。

## 0.3 文科系がんばれ

プログラミングは理科系の人がやればいいと考えている文科系の学生の皆さんも少なくないようです。プログラミングは確かにコンピュータという複雑な機械を操る術ですし、見た目、ソースコードは数式のように見えます。しかしながら、実際のプログラムは「人が行っていること」を「機械に代行」してもらおうということです。ですから人が行っていることをしっかりと把握することがむしろ大事で、この点では文科系の皆さんのがんばりが得意だったりします。これまでの授業では文科系の学生の方も多く履修し合格しています。

## 0.4 本書の構成について

本書は 2018~2020 年度の実践にもとづいて構成されています。このため、前から順に演習していただけるように構成しています。また授業に関連はするものの、少し横道にそれる話題については「コラム」としてとりまとめてあります。

## 0.5 本書での表記

Python については簡単な命令は Python Shell という対話的な環境で 1 行ずつ試してみることで学んでいただき、一方で、まとめたプログラム（スクリプト）はエディタ上で作成し、一括実行することで学ぶという 2 通りの方法を使い分けて学習を進めることができます。

Python Shell で試してほしい機能については、文中で

**a = 1 + 2**

と K2PFE フォントにより赤字で示しました。それを実行した結果については青字で 3

のように示しています。実際に試しながら学習してください。

一方、まとめたプログラムについては、3ないし2列からなる表形式で以下のように示しています。ソースコードの部分をエディタで入力し、実行してください。

行	ソースコード	説明
1	a_=1_+2	
2	print(a)	

なお、以下の点にご注意ください。

- ソースコード中のスペースは空白記号( )に置き換えてあります。
- 通常の Windows フォントでは UTF-8 のバックスラッシュ (\) は円記号(¥)の字形に置き換えられています。K2PFE フォントでは本来のバックスラッシュ(\)の字形となっているので注意してください。

## 0.6 コピペに注意

本書に掲載されているソースコードは Word でのフォーマッティングと PDF への変換を行っているため、PDF の文書では空白の文字数が保存されず、また自動で文字が変わっている場合もあります。PDF からコピーペーストしてもプログラムとしてはエラーになる場合がありますのでご注意下さい。

## 0.7 2020 年度版からの変更

2021 年度版では 2020 年度版での誤植などを修正し、読みにくい文章などを改訂し、2020 年の授業の中で補足した説明などを追記しました。また、2020 年度版では 10 章に置いていたリストの紹介を 4 章に移動し、リストを対象とする for 文の扱いを見直しています。これまでの授業実践に参画いただいた岡本雅子先生に本書の改訂から新たに共著者として加わっていただきました。

また、2021 年度版ではソースコードの記載に新たに開発した K2PFE フォントを用いています。

## 謝辞

2019 年度末に 2019 年度版を公開し、誤植の指摘など本書へのご意見などもいただきました。ご意見をお寄せいただいた方に感謝申し上げます。

また、K2PFE フォントは著者のうち喜多と京都市立芸術大学教授、辰巳明久氏、同非常勤講師、楠麻耶氏、京都大学助教、元木環氏との共同研究により開発いたしました。また、同フォントの開発は、一部、科研費－学術研究助成基金助成金（課題番号 21K02880）の補助を受けて行われました。

本書は CC-BY-NC-ND ライセンスによって許諾されています。ライセンスの内容を知りたい方は <https://creativecommons.org/licenses/by-nc-nd/4.0/deed.ja> でご確認ください。



# 1. コンピュータとプログラミング

---

## 1.1 この章の目的

- プログラミングの対象となるコンピュータの動作の概略とそこでのプログラムの役割について知る。
- プログラミングにおけるプログラミング言語の役割について知る。
- プログラムを作成するさまざまな対象、応用について知る。
- プログラミングの学び方について知る。

### 演習 1-1 この授業の受講動機

以下の質問に答えてください。

1. なぜこの授業に参加しようと思いましたか？
2. プログラミングを学びたい理由は何ですか？
3. なぜ Python 言語を学びたいのですか？
4. プログラミング（の学習）の経験があれば教えてください。有無、どれぐらい？
5. プログラミング（の学習）の経験がある方は、使ったプログラミング言語をお教えください。

## 1.2 コンピュータとプログラム

### 1.2.1 プログラムで動く機械

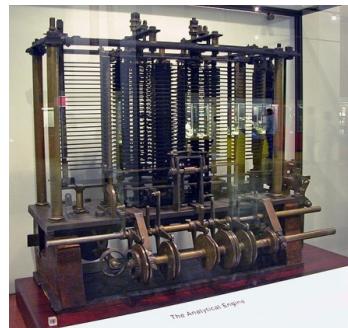
「ジャカード織機」という機械の名前を歴史の授業で聞いたことがあると思います。織物は縦糸の間に横糸を通して織り上げていきますが、どの縦糸を横糸の上にして、どれを下にするかを変えることで「柄」を織ることができます。縦糸の上下についての指示を穴の開いた厚紙（パンチカード、「紋紙」と呼ばれています）で指示できるようにしたものがジャカード織機です。沢山のパンチカードを紐で綴じて順に送ることで複雑な模様を実現します。紋紙をかけ替えることで、別の模様を織りだすことが可能です。

京都の西陣は機織りが地場産業ですが、ジャカード織機がまだ現役で稼働しています。図 1-1

英国の C. バベッジ (Charles Babbage, 1791 – 1871) は機械仕掛けで動く計算装置を開発しようとした人ですが、階差数列から数表を自動生成する機械（階差エンジン）に続いて、ジャカード織機にヒントを得てプログラムで動く機械仕掛けの計算機（解析エンジン）を作ろうとしました。残念ながら、完成には至りませんでしたが、プログラムで計算する機械の先駆けとされています。



ジャカード織機と紋紙  
京都、フクオカ機業にて撮影



バベッジの解析エンジン

図 1-1 ジャカード織機と解析エンジン

[https://commons.wikimedia.org/wiki/File:AnalyticalMachine\\_Babbage\\_London.jpg](https://commons.wikimedia.org/wiki/File:AnalyticalMachine_Babbage_London.jpg)

## 1.2.2 コンピュータの構成要素は電気で動く「スイッチ」

バベッジの時代には複雑な動作の実現には歯車などの機械仕掛けだけが利用可能でした。その後、「電気」で別の「電気スイッチ」を取り切りすることで機械を構成するようになりました。その一つが電磁石で機械的に接点を動かす「**継電器（リレー）**」です。実際、「継電器」でコンピュータを構成することも試みられました。しかしながら、この方式では電気では動くのですが機械的動作を伴うため動作が遅いという欠点がありました。

その後、真空中の電極（陰極、陽極）間を流れる電子をその間においた別の電極に印加する電圧で制御する**真空管**が発明されコンピュータの構成にも応用されました。真空管は電子的動作で速度が速いという利点がありましたが、陰極から電子を放出させるための加熱用にフィラメントを用いていたことから寿命があるという欠点がありました。

次に真空管のような動作を半導体の固体内で実現する素子としてトランジスタが発明されました。トランジスタには寿命が長く、小型で消費電力も小さいという特性があり、多数の個別部品のトランジスタを配線する形でコンピュータが構成されるようになります。

さらに、1つの半導体チップ上に多数のトランジスタやその間の配線などを印刷技術で実現する集積回路が開発され、電子回路の小型化、低価格化を実現します。集積回路技術によりコンピュータの主要部分を1つのチップ上に実現したマイクロプロセッサが開発され、これが現代のパーソナルコンピュータやスマートフォンなど誰でも手にすることができる小型で安価なコンピュータを実現させます。

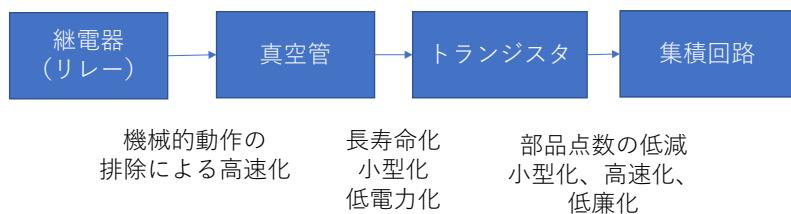


図 1-2 論理素子の進歩

その後は、1つの半導体チップ上に集積できるトランジスタ数（集積度）が40年で100万倍という飛躍的な進歩を見せます。性能が何桁も変わる技術革新は集積回路に加え、記憶装置の容量、通信速度でも達成されています。このような技術革新のおかげで現在ではスマートフォンでYouTubeなどの動画を楽しめるようになりました。

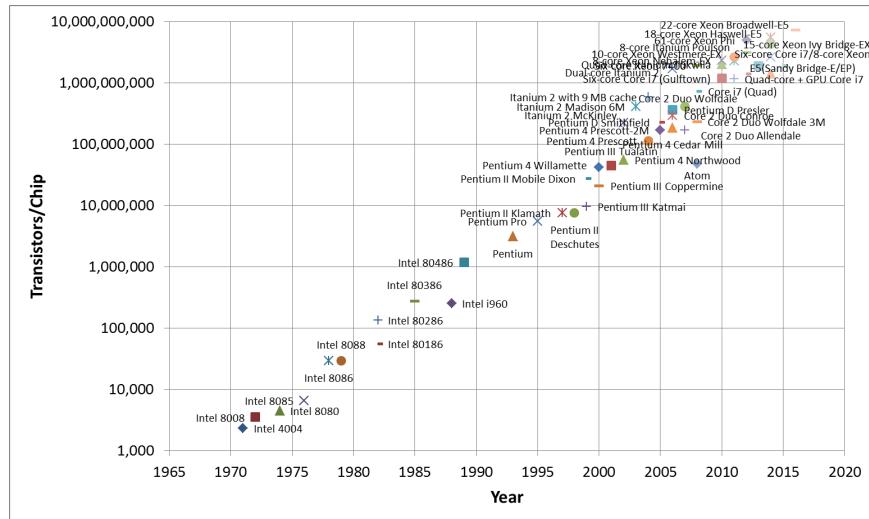


図 1-3 マイクロプロセッサ上のトランジスタ数

[https://en.wikipedia.org/wiki/Transistor\\_count#Microprocessors](https://en.wikipedia.org/wiki/Transistor_count#Microprocessors)

(2017年1月2日アクセス)から Intel 社製のプロセッサを抽出してプロット

### 1.3 コンピュータの仕組み

### 1.3.1 プログラム内蔵方式

現代のコンピュータは複雑な情報処理をどのように実行しているのでしょうか。

コンピュータ（のハードウェア）が1度にできる動作は単純なもので、情報処理の複雑な仕事は、単純な動作の組み合わせをプログラムとして示すことで実現しています。

現代のコンピュータではプログラムは扱うデータとともにメモリ上に格納され、高速に読み出し、実行できるようになっています。このようなコンピュータの構成方式を「プログラム内蔵方式」<sup>1</sup>と呼び、家電製品に用いられる小さなマイコンからスーパーコンピューターまですべてこの方式が採用されています。

させたい仕事に応じてプログラムを切り替えて実行することで同じ（ハードウェアの）コンピュータがさまざまな仕事を行えます。

プログラミングとは実現したい情報処理をプログラムとして記述することです。

<sup>1</sup> 最初期のコンピュータである ENIAC では、計算の設定は「プログラム内蔵方式」ではなく、ケーブルでの配線変更で行われましたが、後継機の EDVAC の開発にあたってこの考え方が提案されました。提案のレポートがフォン・ノイマンによって提出されたことから「ノイマン型コンピュータ」とも呼ばれます。

### 1.3.2 コンピュータの構成と動作

コンピュータのハードウェアの主要な構成要素は CPU とメモリです。

CPU 内には以下のようないくつかの要素があります。

- メモリから命令を取り込み、命令を解析する仕組み
- 実行中のプログラムのメモリ上の番地を示すカウンタ
- データを保持する仕掛け（レジスタ）
- その値に算術や論理演算などをほどこす計算機能（ALU）

コンピュータの基本動作は以下のように単純なものです。

- メモリ上にプログラムと計算に用いるデータが（何らかの手段で）配置されているものとします。
- CPU にプログラムの実行開始場所を与えます
- CPU は以下を繰り返します
  1. メモリ上のプログラムから 1 ステップ分を読み出し、その指示に従って計算、データの転送などを行います。
    - ✧ 計算結果をメモリに書き出す場合もあります。
    - ✧ 入出力を行う場合もあります。
  2. CPU は実行の対象を次の場所に進めます。
    - ✧ 実行場所はプログラムによって変化する場合があります。

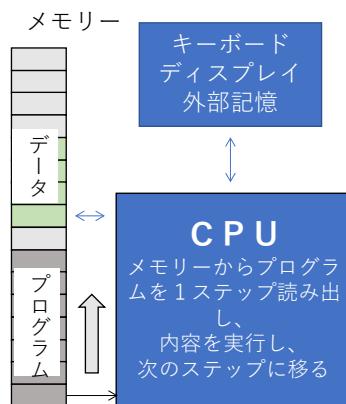


図 1-4 コンピュータ（ハードウェア）の構成

実際のコンピュータのハードウェア(CPU+メモリ) が実行できる命令はハードウェアで高速に処理することを簡単に実現するために極めて単純な命令群として構成されます。このような命令を「機械語」と呼びます。

## 1.4 プログラミング言語

機械語で複雑なコンピュータの応用をプログラミングすることは大変難しい作業です。この問題を解決するために考案されたものがプログラミング言語です。プログラミング言語は次の2つの考え方で成り立っています。

- 機械語よりも数式などに近い形で人間により分かりやすいプログラムを書くためにルールを定める（プログラミング言語の仕様の策定）
- そのルールに従ったプログラム（ソースコードと呼ばれます）を実行するためのプログラムを作成する（プログラミング言語の処理系の構成）

すなわち、プログラミング言語でプログラムを書き、書かれたプログラムは処理系を使って実行するという仕組みです。これは見方を変えれば、「処理系+実際の計算機」で「プログラミング言語で書かれたプログラムを実行できる仮想の計算機」を実現している、ということもできます。

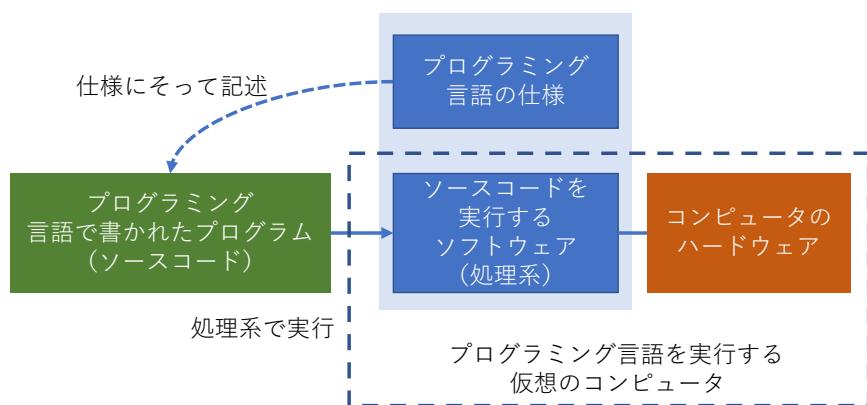


図 1-5 プログラミング言語と処理系

### 1.4.1 さまざまなプログラミング言語

以下の表のようにこれまで、さまざまなプログラミング言語が開発され利用されています。

FORTRAN	COBOL	ALGOL	Pascal	PL/I
BASIC	C, C++, C#	Java	Go	Swift
Perl	Ruby	Python	JavaScript	LISP
Haskell	R	Matlab	ProLog	Scratch

また、プログラミング言語に類似したものとして、Web ページ記述する HTML

やページのスタイルを記述する CSS, データを記述する XML や JSON, データベースへの問い合わせを記述する SQL などが併用されることも多いです。

なぜ、これほど多くのプログラミング言語が考案され、利用されているのでしょうか。なぜ、1つの言語に統一されないのでしょうか。

コンピュータ技術の進展は、開発するプログラムの高度化も求めます。それにともなって、プログラムを効果的に記述するための考え方とそれに基づくプログラミング言語が開発されてきました。また、プログラムをより簡単に書きたい、より高速に、より安全に動かしたいという要望が常に存在しています。特定の用途に適したプログラミング言語へのニーズもあります。これらのことから、新しいプログラミング言語が開発されたり、特定のプログラミング言語の仕様や処理系が改訂されたりしています。

他方で、特定のプログラミング言語で開発されたソフトウェア資産の活用や、その言語での開発を望む技術者はそのプログラミング言語の継続的な利用を求めます。古い言語も、それを捨てることは難しいのです。実際 FORTRAN と呼ばれる科学計算用のプログラミング言語は言語としては最長老ですが、さまざまな改訂も行われつつ現役の言語として使用されています。

プログラミング言語の開発はソフトウェア企業が行う場合や、個人が発案し、コミュニティで発展させる場合などがあります。企業での開発については、プログラミング言語の処理系そのものを有償で販売することを目的とする場合に加え、自社のニーズから開発したものについて、処理系の利用を無償にしたり、ソースコードを公開したりするなどの場合もあります。

## 1.4.2 プログラミング言語の処理系の構成

プログラミング言語で書かれたプログラム（ソースコード）を実際に処理し、実行するためのプログラム（処理系）の構成方式には以下のようにいくつかのものがあります。

### 1) コンパイラ方式

ソースコードを一旦、その内容を実行する機械語に翻訳（コンパイル）し、翻訳された機械語を実行する方式です。コンパイルには手間がかかりますがコンパイルされた実行可能なプログラム（機械語のプログラム）は高速に実行が可能です。

### 2) インタープリタ方式

ソースコードを逐行的に解釈し、動作を模擬する方式。ソースコードの解釈を行

うために実行速度は遅くになりますが、対話的な利用など柔軟性が高い処理系を構成できます。

### 3) 中間コード方式

コンパイラ方式とインタープリタ方式の中間的な方法として、実在する CPU の機械語ではなく、その言語用に想定した仮想の計算機の機械語（中間コード）用にソースコードをコンパイルし中間コードをインタープリタ方式で実行するプログラム（仮想マシン）で実行する方式です。Java や Python で採用されています。

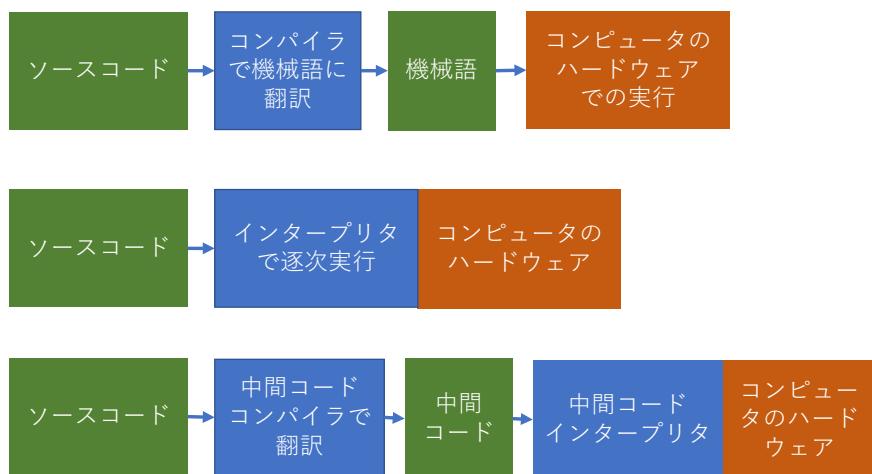


図 1-6 プログラミング言語処理系の構成方式

## 1.5 プログラミング言語 Python

### 1.5.1 Python の歴史

1989 年に Guido van Rossum により実装が始まりました。2000 年に Version 2.0 が公開され、2008 年に Version 3.0 が公開されています。

**注意** Python の Version 3 は Version 2 と上位互換でない (Version 3 が Version 2 の仕様を含んでいない)ため、Version 2 で書かれたプログラムを稼働させるため、現在は両方が併用されています。

**注意** Mac や Redhat Linux では標準で Python が導入されていますが Version 2 がインストールされている場合があります。Version 3 を利用する際には Version 3 の処理系を別途インストールするとともに、どちらの処理系を利用しているのかを確認する必要があります。

## 1.5.2 Python の特徴

- 初心者にも学びやすく、他方で高度なプログラミングも可能です。
- 多様な応用が可能です。
- 科学計算などのライブラリ(numpy, scipy, matplotlib, pandas など)が多くの人により開発されています。
- とりわけ、近年のデータ科学、人工知能（機械学習）技術への関心から、この面でのライブラリが豊富な Python が人気を集めています。

## 1.5.3 Python の配布パッケージ

Python の処理系はいくつか開発されており、これに開発環境やライブラリなどを組み合わせた配布パッケージも複数あります。本授業に関係するものとして、以下の2つを挙げておきます。

- Python : Python の設計者による配布パッケージ、Python の処理系としては C 言語で記述された CPython が使われています。
- Anaconda: CPython に科学計算用のモジュールなどを一括したパッケージ。本授業ではこのパッケージの利用を想定しています。

## 1.6 さまざまな応用

皆さんが Python を学びたい理由として、具体的な応用を想定している人もいるかと思います。Python のさまざまな利用シーンについてみてみましょう。

### 1.6.1 パーソナルコンピュータのアプリケーション

パーソナルコンピュータ上で動くプログラムはその動作環境から大きく2つに分かれます。

- CUI 型の応用プログラム。Windows のコマンドプロンプトなどで動かすプログラムで、キーボードから文字入力を受け取り、画面に文字出力を行う形式のプログラムです。入出力が単純なため比較的学びやすいですが、利用する側には使い勝手が悪くなります。
- GUI 型の応用プログラム。ウィンドウやその上でのボタンなどで操作するプログラムです。利用者には使い慣れた操作を提供できますし、画像などを扱うことも可能ですが、プログラミングすべき事項は多くなります。

ファイルの読み書きやネットワークの操作などは CUI 型でも GUI 型でも共

通の方法でのプログラミングが必要です。

アプリケーションの用途としては、以下のようなものが考えられます。

- 科学計算や数値シミュレーション
- 数値、文字列、画像などのデータの加工、分析
- ゲームやグラフィクスの作品
- Web サイトからの情報の自動抽出（Web スクレイピングと呼ばれています）

## 1.6.2 その他のアプリケーション

- スマートフォンのアプリケーション
- Web サーバなどのネットワーク上のサーバで稼働するプログラム
- 電子回路と連携して動くプログラム。Raspberry PI はこのために開発された Linux や Python が稼働する小型のコンピュータです。

# 1.7 プログラミングの学び方

## 1.7.1 プログラミングが難しい理由

コンピュータのプログラミングはさまざまな理由で難しさを抱えています。どこがなぜ難しいのかを把握しておくことは学習の助けになるでしょう。

### 1) プログラミング言語を構成している概念が分かりにくい

我々が日常、使用している自然言語でも、複雑なことを表現するために文法や語法などのさまざまなルールで運用されています。プログラミング言語も複雑なプログラムをうまく表現するためにさまざまな概念、仕組みが取り入れられています。これらを一度に理解する必要はありません。やさしいものから徐々にステップアップしてゆくことができます。

### 2) エラーへの対応ができない

プログラムではタイプミスや考え方違いなどできまざまなエラー（プログラムに喰う虫に例えてバグと呼ばれます）が発生します。バグは出て当たり前だというぐらいの気持ちで取り組むことが大事ですが、

- ソースコードの文法にそった正確なタイピングが必要であることを理解する必要があります。

- 文法エラーのほか、プログラムについての考え方により、予期した動作をしないこともあります。
- エラーの大半はプログラミングをしている人間に原因があります。どういうエラーかをしっかりと理解し、対処する経験値を上げることが求められます。
- エラー対応は「エラーが生じたという結果」から「エラーを生じさせている原因」を探る、いわば「逆方向」の推論です。結果に適合すると思われる原因についての仮説をさまざまに想定し、実際にそれが原因なのかを検証して行くことが求められます。
- エラーについて、コンピュータの応答（コンピュータが処理できなくなった状況）とプログラムの実際の誤り箇所は異なっていることがあります。

### 3) 実現したい機能をプログラムに展開できない

プログラミング言語を構成する要素を学んでも、それをどのように組み合わせればやりたいことが実現できるのかはなかなか分かりません。鋸や金槌を使えるようになっても、家がどのような構成になっているのかを知らなければ家を建てることはできないのと同じです。

やりたいことを普通の言葉でしっかりと分析し、手順を明確にしたうえで、プログラムを書くことが必要です。比較的簡単な応用例を通して学んでいけばいいでしょう。

### 4) プログラムが複雑になって分からなくなる

プログラムが長くなると、複雑さが増して、何をプログラミングしているのか理解することが急に難しくなります。初学者にとっては 100 行のプログラムでも大変に感じるようですが、長くてもわかりやすいプログラムを書く方法を身に付けながら、100 行程度のプログラムを書くことを目標にするといいでしよう。

### 5) 大きなプログラムには開発手法がある

世の中の巨大なプログラムはソースコードの行数で数億行にもなるそうです。もちろん一人で書くことはできませんし、すべてを頭に入れることもできません。大きなプログラムを書くには、それなりの方法と道具があります。例えば以下のことです。

- プログラム全体をしっかりと設計する
- 分業できるようにモジュールに分割する
- モジュール単位でプログラムを作成し、テストをする。

- 全体を組み上げテストをする。

100行程度のプログラムが書けるようになれば、これらのこと意識して、より大きなプログラムの作成に挑戦されるといいでしょう。

## 1.7.2 プログラミングの学び方

どのようなことでも「学び方を学ぶ」ことは重要です。

- 外国語はどうやって身に着けますか？
- 数学はどうやって身に着けますか？

プログラミングは実際に手を動かしてプログラムを書く技能ですので、その学習は外国語や数学とも似た面があります。ただし、書いたプログラムの動作はその場で確認できますし、面白い応用も可能ですので、外国語や数学ほど恐れなくてよいでしょう。

### 1) 動機付け：興味の持てる課題に取り組むこと。

- 学習の動機は学習を続けてゆくうえでとても大事です。「プログラミングができるようになりたい」と漠然とおっしゃる方も少なくないですが、それだけでは目標設定が曖昧過ぎて動機を失いがちです。難易度の程度はありますが、「ゲームを作ってみたい」などでもいいので、ご自身が興味の持てる課題・応用に取り組むことで学習の動機を維持しましょう。

### 2) プログラミングの学習はたくさん読む、書くが基本です。

- 例題をたくさんタイピングして実行する。
  - 「理解して実行」ではなく、「実行を通じて理解」します。
  - 単語、記号、表記のパターンを覚える。よく使うパターンを全体として身に着けることも重要です。
  - ソースコードのタイピングをより速く、より正確にすることで学習効率が向上します。

### 3) 音読、訳読

声に出して読む、ソースコードの記号を音読することや、日本語で意味を訳して読むことは、指導を受けたりする際の**コミュニケーションを円滑にする**意味からも重要です。

#### 4) ティンカリング：例題のプログラムをいじくって遊ぶ。

- 例題を少し変えて、どのようなことが可能かを探る。
- 複数の例題を組み合わせることに取り組む。これにより、プログラムを組み合わせるにはどのような調整が必要かを理解できます。

#### 5) トレース

- プログラムがどのように実行されるのか自分で解釈しながら追いかけます（トレースと言います）。

#### 6) エラーに対処できるようになること

- プログラミングではエラーへの対処は常に生じます、それ自体が重要な学習目標です。
- 実際のプログラミングでは予期せぬエラーへの対処が求められますが、意図的にエラーのあるプログラムを書いてみて、何が起きるのかを知ることで経験値を高めることも効果的です。
- エラー時のメッセージを読む。プログラムのエラーが出てもメッセージを読まない初学者の方をよく見かけます。エラーメッセージはどうしても難解に思えますが、エラーが生じた場所やエラーの種類を示してくれています。意図的に生じさせたエラーならメッセージの理解も容易なはずですのでエラーメッセージに慣れてください。

#### 7) 情報探索

- 以下のような、内容や方法でプログラミングのための情報探索ができること
  - プログラミングのためのより高度な概念やライブラリの使い方を学ぶ。
  - プログラミングを支えるツールを知る。
  - 書籍やネット上の情報などを探せる。
  - 分からないときは人に聞く、これは聞くことができるだけのコミュニケーション力も必要になります。

### 1.7.3 プログラムで使う文字

私たちはプログラムの中で日本語を含めた文字を扱いたいですが、他方で多くのプログラミング言語が英語を基礎に設計されているため、文字と文字コードに対するある程度の理解と注意が求められます。コラム「プログラムと日本語一終わりそ

うで終わらない文字コードとの闘い」も参照してください。

- Python(を含めた多くのプログラミング言語)の基本は半角の英数字です。
- 全角文字は「文字列としてのデータ」と「コメント」だけで使います<sup>1</sup>。
- Pythonでは大文字と小文字を区別します(Case Sensitiveと言います)。
- プログラミングでは、さまざまな記号も使います。
  - キーボード上の位置だけでなく、
  - 他の方とのコミュニケーションのために「呼び方」も覚えることが必要です。
- このほか、Ctrlキーを押しながら、例えばCというキーを押すなどのキー操作も求められます。この場合Ctrl-Cなどと記述します。Altキーについても同様です。



図 1-7 JIS キーボードの配置

英文用キーボード(ASCII配置)では記号などの配置が異なっています。

以下の表に主な記号やその読み方、使用上の注意をまとめておきます。この表は文献[1]から著者の了解を得たうえで、一部補足して作成したものです。

表 1-1 プログラミングで用いる記号と読み

記号	読み方	注意
□	空白、スペース	記号欄では分かりやすくするため□と表記しました。 プログラムには半角スペースを用います、日本語文字列として用いる以外の全角スペースは見分けにくいエラーになるので注意。

<sup>1</sup> Pythonでは変数名に漢字も使用可能ですが、そうではないプログラミング言語も多いので避けたほうが無難です。

!	感嘆符, エクスクラメーションマーク	
"	二重引用符, ダブルクオーテーション, ダブルクオート	Python では文字列を "もしくは 'で囲みます。どちらでも構いませんが開始と終了に同じ引用符を使ってください。
'	一重引用符, アポストロフィ, シングルクオーテーション	
#	シャープ, ナンバー	
\$	ドル, ダラー	
%	パーセント	
&	アンド, アンパサンド	
*	アステリスク, アスタリスク	
+	プラス	
,	カンマ, コンマ	
-	マイナス, ハイフン	
.	ピリオド, ドット	
/	斜線, スラッシュ	
:	コロン	これらの違いに注意
;	セミコロン	
<	小なり, 左不等	
>	大なり, 右不等号	
=	イコール, 等号	
?	クエスチョンマーク, 疑問符	
@	アット, アットマーク, 単価記号	
¥\	円記号 逆スラッシュ	JIS コードでは \ と同じコードに¥を割り付けています。Python で用いる unicode (UTF-8) ではこれらは異なるコードになっていますが、Windows の多くのフォントでは \ の代わりに ¥ の字体が収められています。mac ユーザは素直に逆スラッシュを使ってください。
^	やま, アクサンシルコンフレックス	
_	アンダーライン, アンダースコア, 下線	Python ではアンダースコアを 2 文字続けて __ のように使いことが多いです。
	縦線, 縦棒	
~	なみ, チルド, チルダ	
[	大括弧, 角括弧 (開く)	Python を含む多くのプログラミング言語では、さまざまな意味で括弧を使い分けます。タイプミスが発生しやすいです。
]	〃(閉じる)	
{	中括弧 (開く)	
}	〃(閉じる)	
(	小括弧, 丸括弧, 括弧 (開く)	
)	〃 (閉じる)	
<=	小なりイコール	2 文字です
>=	大なりイコール	2 文字です
!=	ノットイコール	2 文字です

==	イコールイコール	2 文字です
----	----------	--------

### 演習 1-2 プログラミングで使う記号

プログラミングで使う記号のキーボードでの配置と読みを確認してください。

## 1.8 プログラムを構成する基礎的な概念

Python によらず広くプログラミング言語を学習する際には以下のようない事項がプログラムを構成する基礎的な概念になります。

- 算術、文字列、論理（真偽）の演算
- 変数、変数への代入、変数の値の評価（代入されている値を使うこと）
- 条件判断によるプログラムの実行の切り替え（分岐）
- プログラムの特定箇所の繰り返し実行
- 定型動作の記述と呼び出し（関数の定義と呼び出し）
- 複合的なデータの取り扱い
- 入出力（端末、GUI、ファイル、ネットワーク）

## 1.9 プログラムの「どこ」を作るか

現代ではアプリケーションプログラムをすべて自身で作成することはめったにありません。以下の 2 つの中間に位置する部分を用途に応じて作成するのだと理解してください。

- フレームワーク：パーソナルコンピュータ上の GUI を用いるアプリケーションや Web サーバ上のアプリケーションでは、GUI 全体や Web サーバそのものは事前に用意されているものを使います。このようなプログラム全体の大枠をフレームワークと呼びます。
- ライブラリ：他方でプログラムの中で、sin や cos といった数学関数などのように誰もが部品として使いたいものはライブラリとして用意されているものを使います。

すなわちフレームワーク上で、アプリケーション固有の動きを、ニーズにあったライブラリを活用しながら自身でプログラミングすることになります。

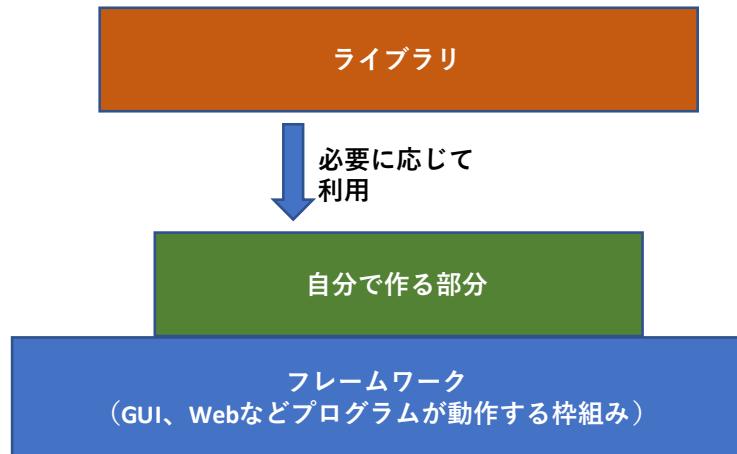


図 1-8 フレームワークとライブラリ

## 参考文献

- [1] 喜多 一, 岡本雅子, 藤岡健史, 吉川直人: 写経型学習による C 言語プログラミングワークブック, 共立出版 (2012)

## 2. Python の実行環境と使い方

---

### 2.1 本章の学習の目標

- Python の統合環境 IDLE を起動できる。
- IDLE 上での Python シェルを操作できる。
- IDLE で Python のプログラム（スクリプト）を編集するエディタを操作できる。

### 2.2 学習環境の想定

本書では京都大学の教育用コンピュータシステムの固定端末での学習を前提に以下の環境での Python の学習を想定しています。

- 基本ソフトウェアとして Windows 10
- Python の配布パッケージとして Python バージョン 3 で構成された Anaconda
- Python の統合開発環境として Anaconda に含まれている IDLE

個人所有の PC についてはそれぞれ Anaconda をインストールしてください。なお、11 章で紹介するする数値計算などのモジュール numpy, matplotlib, pandas を用いない場合は、本家の Python パッケージでもかまいません。

Python の開発環境としては IDLE の他にも Jupyter Notebook や Spyder などさまざまなものがあります。本書で統合開発環境として IDLE を用いる理由は機能が限定されていて初学者には分かりやすいこと（指導者が指導しやすいこと）と、例として取り上げるタートルグラフィックスを稼働させやすいことが理由です。ただし、Windows と macOS で起動方法や動作が若干異なる点はご留意ください。macOS での利用については後で要点をまとめています。

## 2.3 準備

この授業で作成する Python のプログラム（スクリプト）を保存するフォルダを作成してください。例えばドキュメントの下に「Python Scripts」というフォルダを作成します<sup>1</sup>。

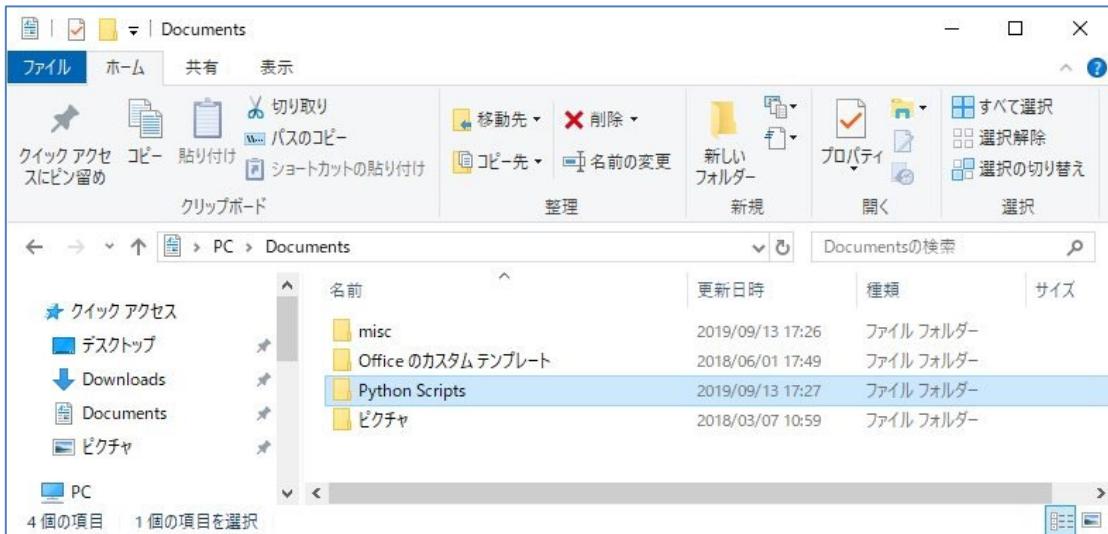


図 2-1 プログラム保存用フォルダの作成

## 2.4 IDLE の起動

スタートメニューから Anaconda3 というフォルダの中の Anaconda Prompt を選んでダブルクリックで起動してください。起動したらこのウィンドウ内で `idle`（大文字、小文字は問いません）と入力し ENTER キーを押すことで IDLE が起動します。

<sup>1</sup> 教育用コンピュータシステムでは NextCloud というサービスとして稼働させている N: ドライブにフォルダを作成すれば、自宅の PC など外部からもアクセス可能です。

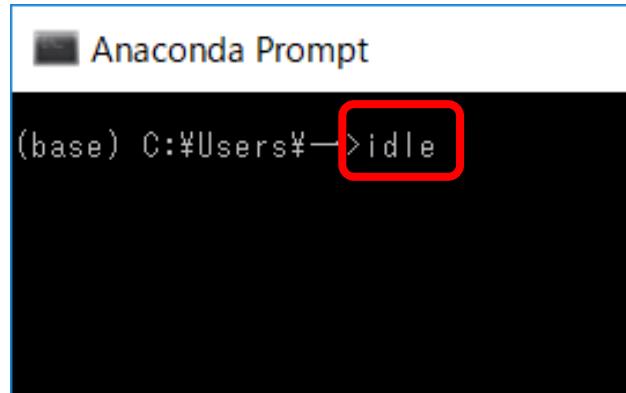


図 2-2 Anaconda Prompt からの idle の起動

## 2.5 Python シェル

### 2.5.1 起動の確認

IDLE を起動すると図 2-3 のような Python シェルが現れます。これは Python を対話的に実行する環境です。次の 2 点を確認してください。

- ウィンドウのタイトルの確認。ウィンドウのタイトルに実行している Python のバージョン（ここでは IDLE Shell 3.8.11 となっています。）が表示されます。複数の Python のバージョンがインストールされている場合、誤ったバージョン（Python バージョン 2 とか）が起動されことがあります。その場合、IDLE の起動方法などを確認してください。
- プロンプトの確認。ウィンドウの中の「>>>」が入力を促進する記号（プロンプト）です。これに続けて Python の命令をキーボードから入力できます。

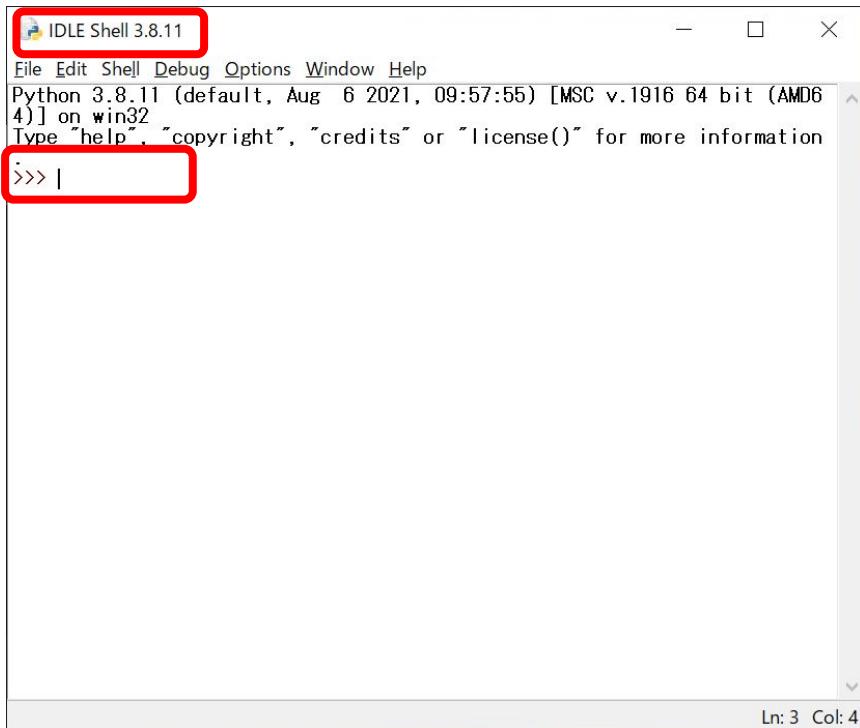


図 2-3 IDLE の Python シェル

### 2.5.1 Python の命令の実行

Python シェルのプロンプトに続けて

**1+2**

と入力し、ENTER キーを押してみてください。以下、**シェルから入力するものは赤字**で書きます。これは「1+2 という計算をする」立派な Python のプログラムです。シェルはこれを実行して

**3**

と答えてくれるはずです。以下、**実行結果は青字**で書きます。

Python での算術演算は下の表のようになっています。掛け算には「\*」、割り算には「/」を用います。数学での計算と同様、掛け算、割り算は足し算、引き算より優先されます。また計算順序を優先するために()が使えます。

なお Python version 3 では「/」は整数同士の割り算でも結果が float 型になります。整数商が必要な場合は「//」を使ってください。またプログラムでは割り算の「余り」を使いたいことも多いのですが、このための演算子「%」が用意されています。

表 2-1 Python の算術演算

演算子	演算	備考
+	足し算	
-	引き算	
*	掛け算	
/	割り算	Python では結果は float 型
//	整数の割り算	
%	剰余	割り算の余りを求めます
**	べき乗	2 文字です。
( )	演算の優先	他の括弧は使えません、

## 演習 2-1 算術演算の確認

Python シェルで算術演算を練習してください。

次に以下の 2 行を（一行ずつ）入力してみてください。右図参照

a = 1 + 2

a

1 行目は等号「=」の左辺の「a」という変数に右辺の「1+2」という式の計算結果を代入しなさい、という命令です。この行の実行ではシェルは何も表示せず次の入力を要求します。

2 行目の実行で変数 a の値を確認しています。

3

という結果が表示されているはずです。次に

print(a)

という命令を入れてみてください。print() は () 内の式を文字としてシェルに出力する関数です。やはり

3

と表示されます。

Python Shell では変数名だけ入力すると、その値が表示されます。これに対して次に述べるプログラムをまとめて記述して一括実行するスクリプトでは明示的に

```

IDLE Shell 3.8.11
File Edit Shell Debug Options Window Help
Python 3.8.11 (default, Aug  6 2021, 09:57:55) [MSC v.1916 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license()" for more information
>>> a = 1 + 2
>>> a
3
>>> print(a)
3
>>> |

```

図 2-4 Python Shell での操作

`print()` 関数などを使わなくてはなりません。

## 2.6 スクリプトの作成と実行

次に複数行の Python の命令を書いて、まとめて実行する方法を学びます。このためには命令を編集する IDLE Editor を使います。

### 2.6.1 新規ファイルの作成

新しいプログラムを作成するのでシェルウィンドウの「File」メニューから「New File」を選びます。IDLE Editor が起動されます。

### 2.6.2 IDLE Editor の確認

IDLE Editor と Python Shell はよく似ています。間違わないように以下の 3 点を確認してください。

- Window のタイトルは編集しているファイル名になります。New File を選んだ場合は「Untitled」になっています。
- Window のメニューは Python Shell と異なっています。「run」というメニューあることを確認してください。
- Window 内は空白です。シェルのプロンプト「>>>」は表示されません。
- Windows の右下にカーソルのある行と列が表示されます。

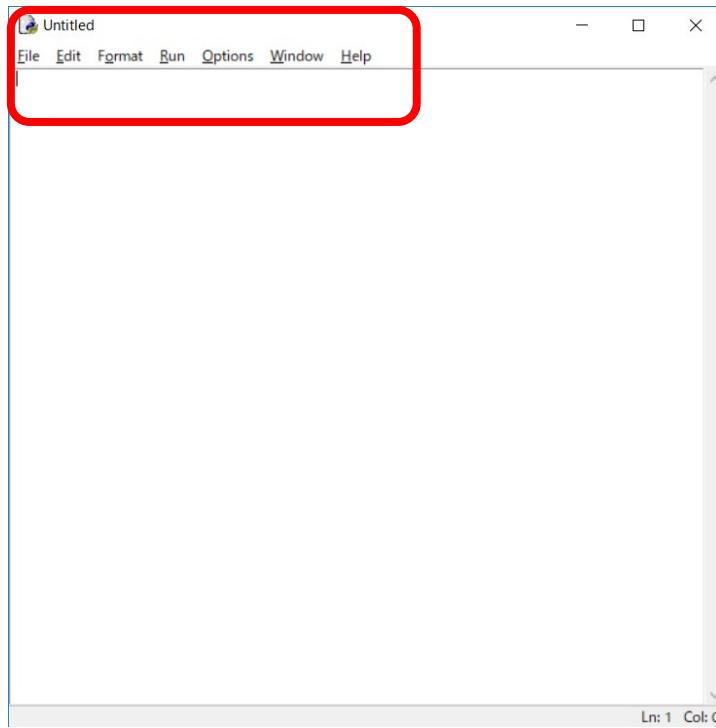


図 2-5 IDLE Editor, Python シェルとの違いに注意

### 演習 2-2 IDLE の Python Shell と Editor の違いの確認

IDLE の Python Shell と Editor の違いを確認してください。

### 2.6.3 Python プログラムの記述, 保存, 実行

2 行だけのプログラムですが IDLE エディタで下の表の黄色の部分を入力してください。

#### プログラム 2-1 (p2-1.py)

行	ソースコード	備考
1	a = 1 + 2	右辺 1+2 の計算結果を左辺の変数 a に代入する。
2	print(a)	変数 a の値を画面に出力する。

タイプミスがないことを確認して、メニューの「Run」から「Run Module」を選びます。Run はプログラムの実行、モジュールは現在、編集中の Python プログラムを意味します。

新規のプログラムなので保存する必要があります。Python のプログラムを置くフォルダに p2-1.py (.py は Python プログラムの拡張子) という名称で保存を指示すると、フォルダにプログラムを保存した後、Python Shell 上で実行され、実行結果が表示されます。

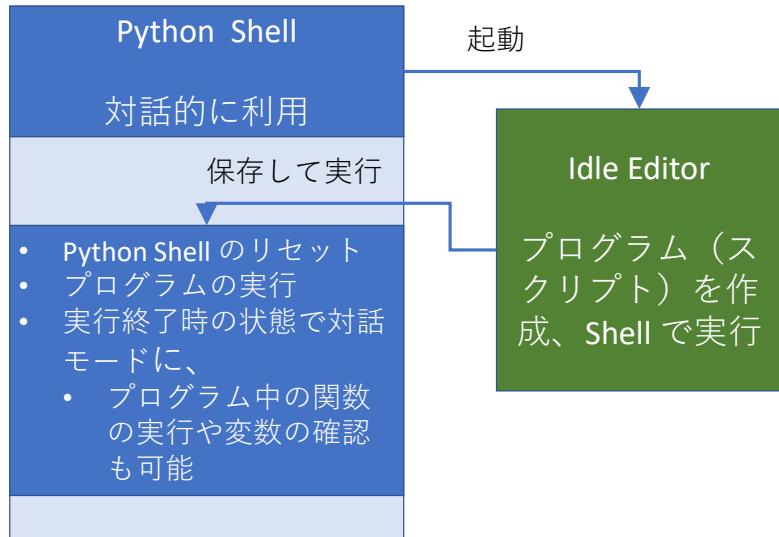


図 2-6 IDLE での Shell とエディタの連携

Idle Editor から実行を指示すると、プログラムがファイルに保存され、Python Shell がリセットされてプログラムが実行されます<sup>1</sup>。実行が終了すると Python Shell は終了した状態でキーボードから入力を受け付ける対話モードになります。これにより、プログラム中で使った変数の値を確認したり、プログラム中の関数を呼び出したりすることができます。

#### 演習 2-3 p2-1.py の実行後の確認

p2-1.py の実行が終了した時点で以下のコマンドを実行して変数 a の値を確認してください。

```
print(a)
```

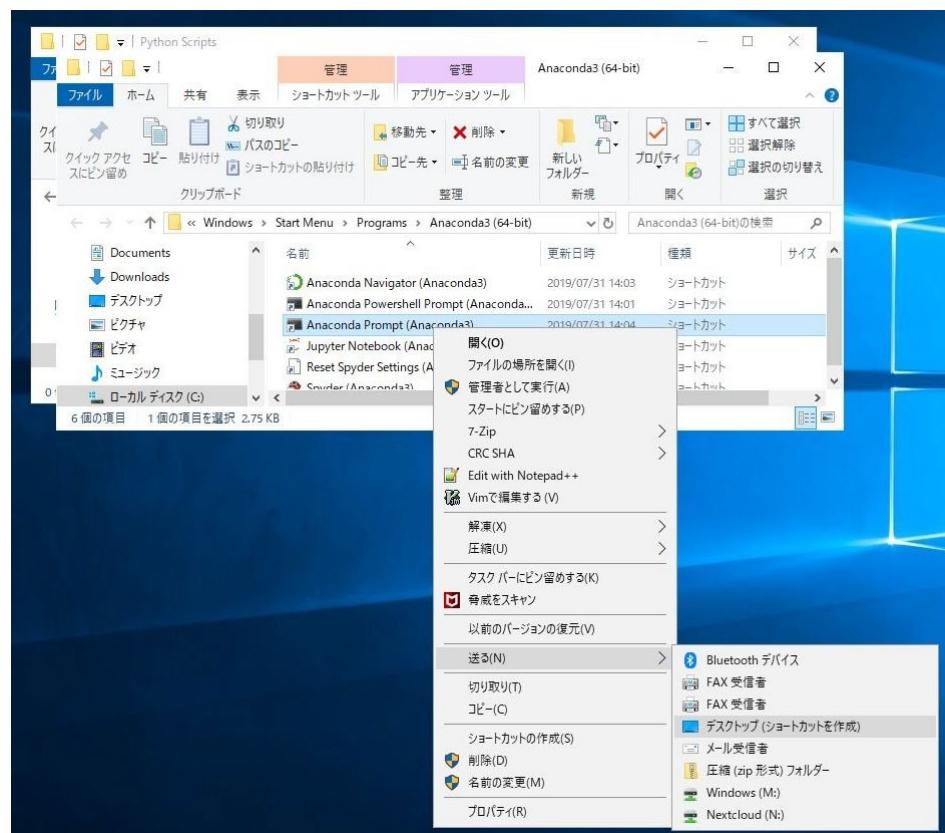
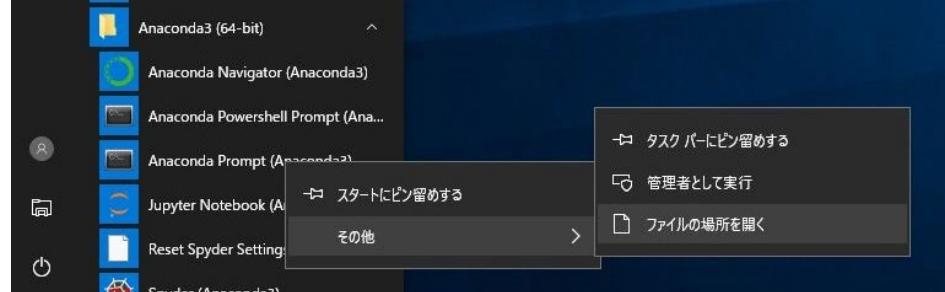
## 2.7 Anaconda Prompt での作業フォルダの設定

Python のプログラムを入れるフォルダを作成したのでこのフォルダを最初に開くように設定します。IDLE では作業フォルダを指定することができないので IDLE を起動する Anaconda Prompt の作業フォルダを指定することにします。以下の手順で設定してください。

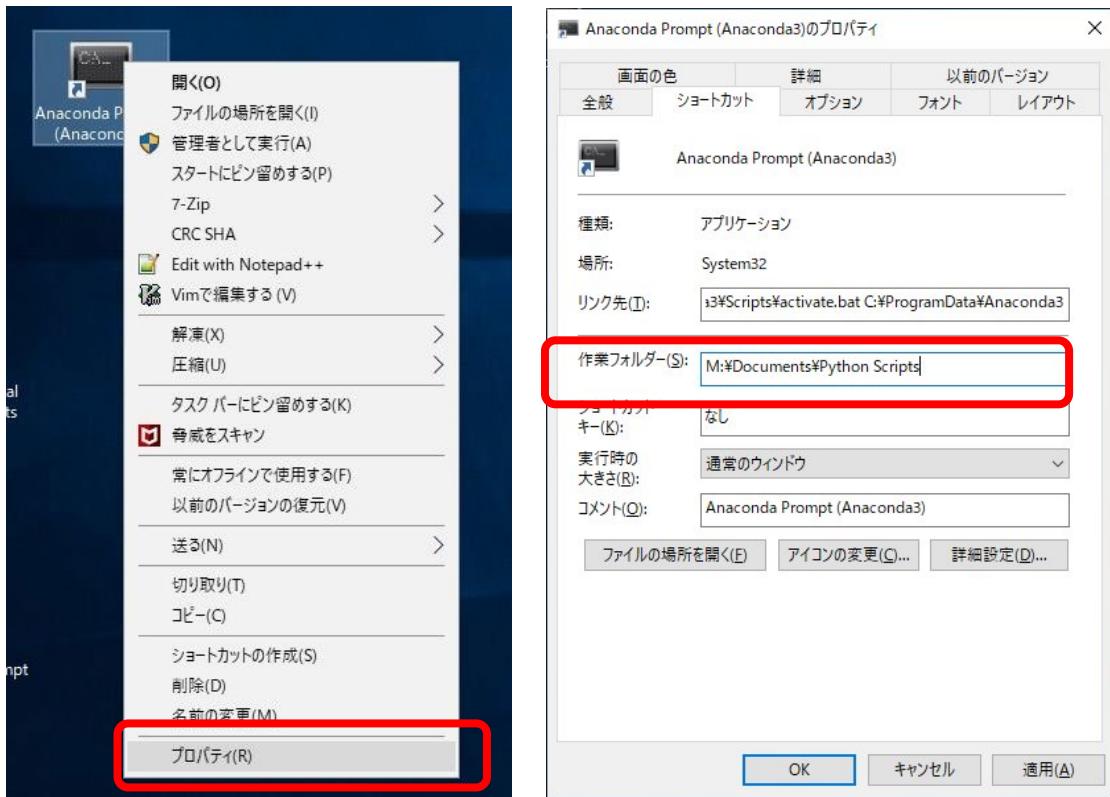
### 1. デスクトップに Anaconda Prompt のショートカットを作成

<sup>1</sup> Shell のリセットが必要なのは、それまでに対話的に使用してきた変数などが残されているので、その影響をなくすためです。

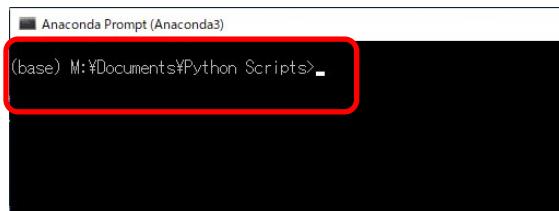
1. スタートメニュー Anaconda Prompt で右クリック
2. 「その他」→「ファイルの場所を開く」を選択
3. エクスプローラの Anaconda Prompt アイコンで右クリック
4. 「送る」→「デスクトップ（ショートカットの作成）」を選択。



2. デスクトップ上の Anaconda Prompt ショートカットに作業フォルダを設定
  1. デスクトップの Anaconda Prompt アイコンを右クリック
  2. 「プロパティ」を選択
  3. 「作業フォルダ」に Python のスクリプトを置くフォルダを設定
  4. 「OK」ボタンをクリック



以後、デスクトップのアイコンをダブルクリックすれば Anaconda Prompt や idle が設定された作業フォルダで動きます。



## 2.8 IDLE のキー操作など

IDLE の Python Shell や IDLE Editor はシンプルなものですぐ、いくつか便利なキー操作が設定されています。IDLE のオンラインマニュアルでも読めますが、よく使うものを 16 章の「IDLE Python 便利帳」にまとめておきました。

## 2.9 拡張子の表示

ファイルの種類は拡張子（ファイル名の末尾のピリオド以下につけられた名前、Python のプログラムの場合は「.py」）で示すことになっています。プログラミングを行うとさまざまな形式のファイルを操作しますが、Windows のエクスプローラー

では標準ではファイルの拡張子を隠しています。

エクスプローラーの表示タブで下図の設定を行えばファイルの拡張子を表示できるようになります。

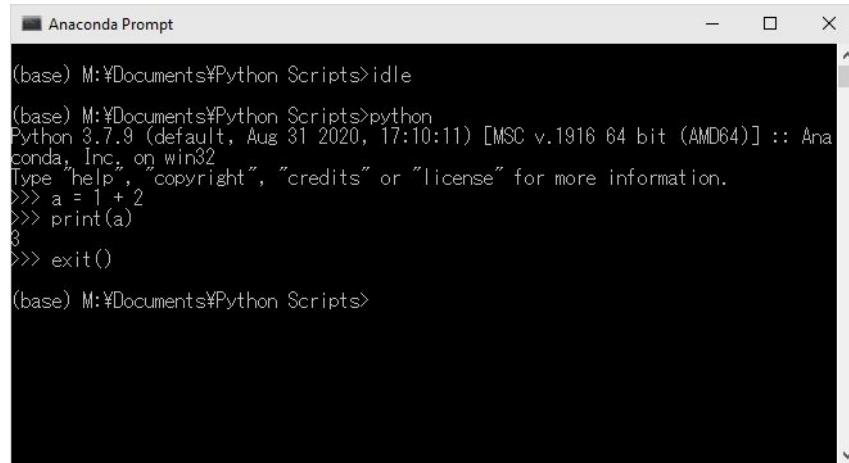
**注意：**ファイルの拡張子を表示できるようにすると変更も可能になります。うっかり拡張子を変更するとアプリケーションとの関連付けができなくなるので注意してください。



## 2.10 Python コマンドの実行

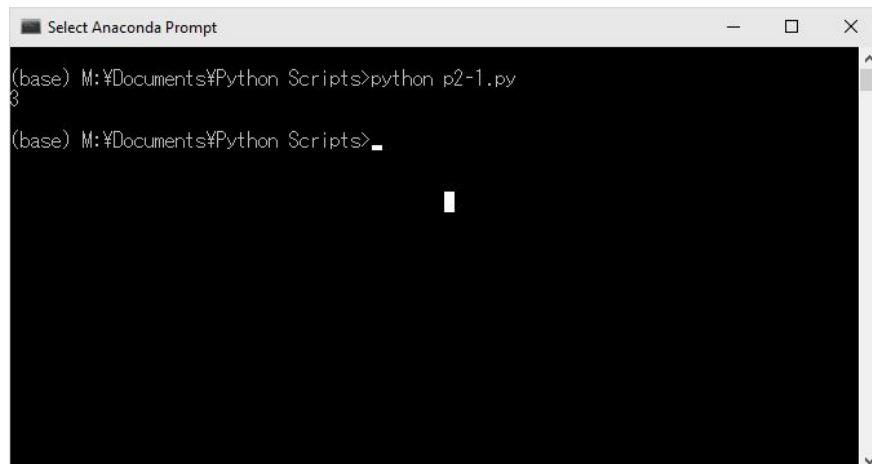
Python のプログラムは Anaconda Prompt 上で直接、実行することも可能です。以下のことを確認してください。

1. すでに開いている IDLE や Anaconda Prompt は一旦、終了してください。
2. 前節で示した作業用フォルダの設定を行います。
3. デスクトップ上のショートカットから Anaconda Prompt を起動します。
4. cd と入力して ENTER キーを押し、作業用フォルダが正しく設定されていることを確認してください。
5. dir と入力し、ENTER キーを押すとフォルダにあるファイルの一覧が表示されます。p2-1.py が含まれていることを確認してください。
6. python と入力すると anaonda prompt で python シェルが起動することを確認してください。今は起動を確認するだけでよいので exit() と入力するか、CTRL キーを押しながら C を入力(Ctrl-C)してシェルを終了してください。
7. python p2-1.py と Python プログラム（スクリプト）名を指定して python コマンドを実行するとプログラムが実行されます。実行結果を確認してください。
8. python -i p2-1.py と -i オプションを指定するとプログラム実行後に対話モードを継続します（IDLE での実行と同様）



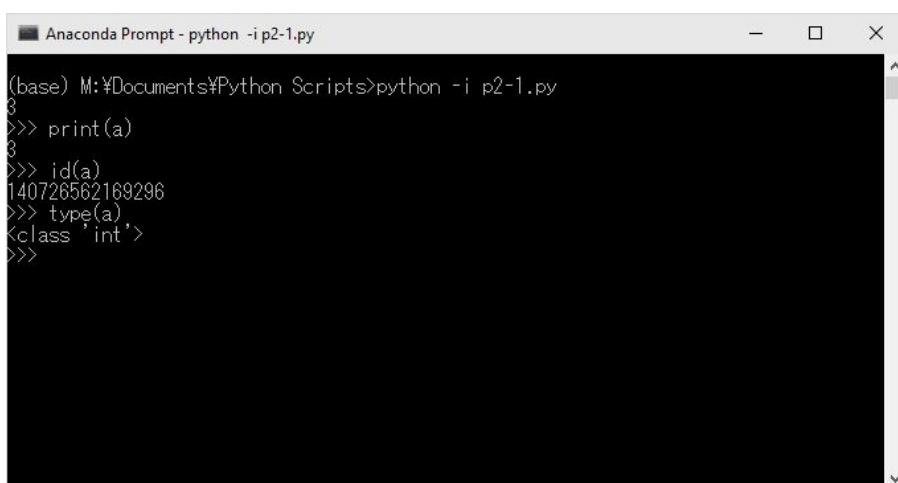
```
(base) M:\Documents\Python Scripts>idle  
(base) M:\Documents\Python Scripts>python  
Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Ana  
conda, Inc. on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> a = 1 + 2  
>>> print(a)  
3  
>>> exit()  
(base) M:\Documents\Python Scripts>
```

図 2-7 対話モードでの Python の起動



```
Select Anaconda Prompt  
(base) M:\Documents\Python Scripts>python p2-1.py  
3  
(base) M:\Documents\Python Scripts>
```

図 2-8 スクリプトを指定した Python の実行



```
Anaconda Prompt - python -i p2-1.py  
(base) M:\Documents\Python Scripts>python -i p2-1.py  
3  
>>> print(a)  
3  
>>> id(a)  
140726562169296  
>>> type(a)  
<class 'int'>  
>>>
```

図 2-9 -i オプションでスクリプト実行後に対話モードを継続

## 2.11 Python を学ぶ環境づくり

- Python の処理系：自分の PC に Anaconda をインストールする。
- Python のリファレンスマニュアルの確認
- Python の本（自分に合いそうなものを、1冊は手元に）
- 英語の辞書（資料の確認、変数や関数の命名）
- ノート、筆記具（PC 上のツールでも可）：気づいたことを書き留める

演習 2-4 あなた自身が Python を学ぶ環境を用意し、報告してください。

演習 2-5 本日の演習内容をご自身の学習環境で再確認してください。

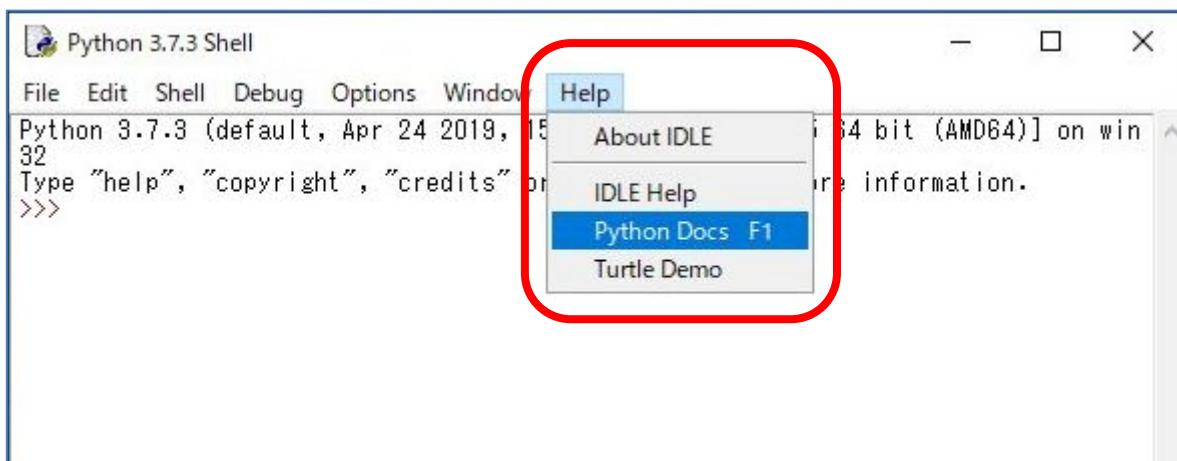


図 2-10 IDLE からオンラインマニュアルの起動

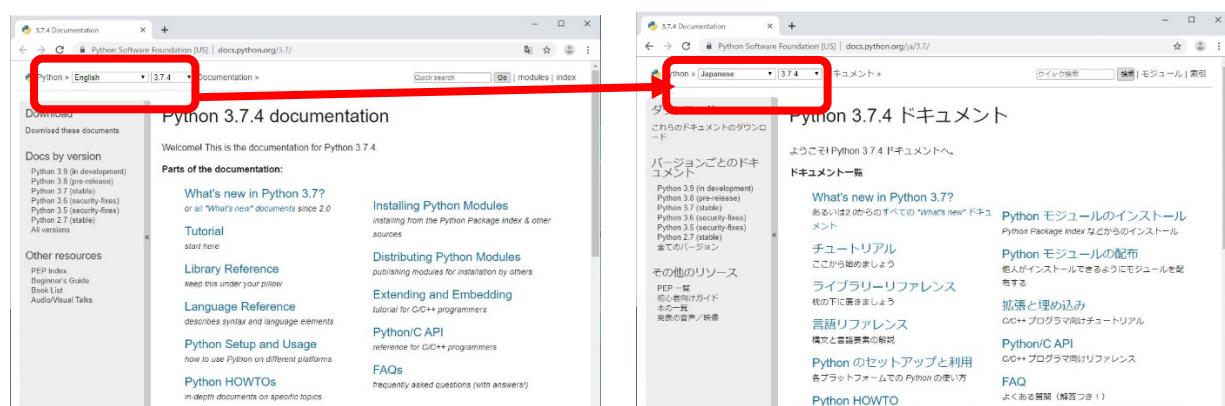


図 2-11 Python のオンラインマニュアル（右は日本語を指定した場合）

## 2.12 Mac ユーザへ

本書では Windows 環境での Python の利用を説明しますが、Mac での利用にはいくつか課題があります。以下を参考にしてください。

### 2.12.1 Mac での Python の導入と IDLE の起動

#### 1) Anaconda のインストール

以下サイトで「download」ボタンを押すか、ページを下にスクロールして、表示された画面で「Mac OS」の Installer をダウンロードしてください（どれを選ぶかわからない場合は 64-Bit Graphical Installer を選択してください）。

<https://www.anaconda.com/products/individual/>

ダウンロードしたパッケージファイルを実行し、インストールします（途中のインストール方法の選択がよくわからない場合は「自分専用にインストール」をクリックして選択する）。

新しい CPU である M1 を搭載した Mac では古い CPU 用のプログラムを実行するためのソフトウェア Rosetta のインストールを尋ねられますので、これもインストールしてください。

なお、Anaconda をインストールするとターミナル環境がカスタマイズされ Python 実行に適した Conda 環境になります（入力時の各行頭に「(base)」と表示された状態）。プログラミング演習のみにターミナルを利用する場合はこれまで問題ありませんが、ターミナルで他のソフトを実行する時に問題が出たなどで環境を元に戻したい場合は「conda deactivate」と入力してください。戻した後で再び Conda 環境を利用したい場合は「conda activate」と入力することで利用可能です。

ターミナルに Conda 環境がインストールされない場合 ((base) が表示されない場合) はターミナル環境で以下を実行し、一度、ターミナルを閉じて、再度開いてください。最後の zsh はターミナルで稼働するシェルの名前です。zsh 以外のシェルが使われている場合はそのシェル名にします。

```
/opt/anaconda3/bin/conda init zsh
```

#### 2) IDLE の起動

Windows のコマンドプロンプトにあたるのはターミナルです。IDLE はターミナルから起動します。

Finder の「アプリケーション」→「ユーティリティ」→「ターミナル」で起動できます。

Anaconda をインストールした状態であれば、「idle3」とキーボードで入力しエンターキーを押せば Python 3 用の IDLE を起動できます(前の節の Windows 版のコマンドプロンプトの場合の「idle」と起動時の名前が違うので注意)。

### 3) IDLE の操作

IDLE Editor を起動するなどでメニュー操作を行う場合に、メニューがクリックできないことがあります。その場合、他のウィンドウやデスクトップ背景などをクリックしてから再び IDLE のウィンドウ（「Python3.8」というタイトルになっています）をクリックするとメニューが操作できるようになります。

### 4) IDLE での日本語入力

日本語入力をする場合、入力が不安定になったり遅くなったりすることがあるので注意してください。また、現時点(2020/9/29)で日本語入力に「Google 日本語入力」を使うと上手く文字が入力されないことがあるため、その場合は Mac の標準の日本語入力を使うことをお勧めします。

コメント等の動作に影響しない部分の文章については、最初から日本語を使わないのも一つの手段です。

### 5) IDLE でのバックスラッシュの入力

この教科書では 11, 12 章でプログラム中にバックスラッシュ「\」(Windows の標準的なフォントでは円記号「¥」)を入力する箇所がありますが、Mac ではバックスラッシュ「\」を入力してください。

バックスラッシュ記号「\」の入力方法ですが、Mac では 左下の「option」キーを押しながら「¥」キーを押すと入力できます。



図 2-12 Mac での逆スラッシュの入力

Option キーを押しながら ¥ キーを押す。

## 6) IDLE の終了

idle を終了したら、ターミナルのウィンドウを閉じても大丈夫です。

### 2.12.2 Mac での Tkinter の問題

上記のほか GUI 環境の Tkinter は Tcl/Tk というパッケージを利用しておる、これに伴う問題や、グラフ描画モジュール matplotlib での日本語フォントの取り扱いなど OS に依存する部分の操作に Windows との違いがあります。

## 参考文献

Python に関する書籍は近年、数多く出版されていて、どれを買っていいのか迷うかと思います。以下、いくつか挙げておきます。文献[2]～[6]は Python の入門書です。文献[7]は名称からは分かりにくいですが Python を解説しながら独学でプログラマーになった著者の経験なども紹介しています。文献[8]はプログラミングの周辺で入門書にはあまり書かれないと解説しています。文献[9], [10]は応用を意識した本です。文献[11], [12]は本書で少し触れる数値計算など学術利用関連のライブラリである NumPy, matplotlib, pandas の解説書です。このほかの書籍を図書館などで探される際には Python Version 3 を扱っていることを確認してください。

- [2] Bill Lubanovic 著, 鈴木 駿 監訳, 長尾 高弘 訳: 入門 Python3 第2版, オライリー・ジャパン (2021)
- [3] 柴田淳: みんなの Python 第4版, SBクリエイティブ (2017)
- [4] 大津真: 基礎 Python, インプレス (2016)
- [5] 松浦健一郎, 司ゆき: はじめての Python エンジニア入門編, 秀和システム (2019)
- [6] 大澤文孝: いちばんやさしい Python 入門教室, ソーテック社 (2017)
- [7] コーリー・アルソフ著, 清水川貴之訳: 独学プログラマー, 日経BP (2018)
- [8] 増井敏克: 基礎からのプログラミングリテラシー, 技術評論社 (2019)
- [9] 日経ソフトウェア編: いろいろ作りながら学ぶ! Python 入門, 日経BP(2019)
- [10] Al Sweigart 著, 相川愛三訳: 退屈なことは Python にやらせよう, オライリー・ジャパン (2017)
- [11] Wes McKinney 著, 瀬戸山ほか訳: Python によるデータ分析入門, オライリー・ジャパン
- [12] Jake VanderPlas 著, 菊池彰訳: Python データサイエンスハンドブック, オライリー・ジャパン (2018)

## 3. 変数と演算, 代入

---

### 3.1 本章の学習の目標

- Python のプログラムでの「実行の流れ」と「情報の流れ」を理解し、順次実行について知る。
- プログラムでの変数の命名と代入、評価について知る。
- Python の基本的なデータ型を知る。
- データ（オブジェクト）の型を調べる `type()` 関数とオブジェクトの所在を調べる `id()` 関数について知る。

### 3.2 プログラムの実行の流れと情報の流れ

#### 3.2.1 順次実行

前章の例

```
a = 1 + 2
print(a)
```

ではプログラムは上から順に 1 行ずつ実行されて行きます。これは「順次実行」といい、プログラムの基本です。このほか、

- 条件によって実行する箇所を切り替える分岐
- 同じ処理の繰り返し
- 関数を呼び出すことで、処理を関数の定義に移すこと

などがあり、これらについては後の章で説明します。Python プログラムのソースコードは「実行の流れ」に沿って書かれています。

#### 演習 3-1 プログラムと楽譜

コンピュータのプログラムは音楽の楽譜と似た面があります。音楽の楽譜も基本は前から順に音符を演奏することです(プログラムは上から下、楽譜は左から右と方向は異なります)。このほか、演奏箇所を切り替えたり、繰り返したりする記法があることを確認してみてください。

The musical score is in 4/4 time, treble clef, and key signature of one sharp. It features lyrics in Japanese and Romanized form. The lyrics are:

(一) ココノヘニハナゾニホヘルセシネンノミ  
(二) みどりふくくすのはかぜにときのかねみつ

At the top right, there is a play button icon labeled "Sound". To its right, the text reads "水梨弘久 作詞" (Music by Hirokazu Mizuri) and "下總院一 作曲" (Composition by Ichiro Shimotsuzawa).

楽譜の例：京都大学の学歌、以下から抜粋

<https://www.kyoto-u.ac.jp/ja/about/operation/symbol/song-a.html>

### 3.2.2 変数を通じた情報の流れ

一方、プログラムでは情報をプログラムの各ステップで加工して行きますが、情報は変数に代入された数値や文字列として扱われます。このため、「実行の流れ」と比較して「情報の流れ」は同じ変数への代入や参照を通じて行なわれるため「分かりにくい」、ということを意識しておいてください。例えば上の例で 1 行目で設定された変数 `a` の値が 2 行目の `print` 関数で使われています。

では、以下のような例ではどうでしょうか

```
a = 1 + 2
a = 3 + 4
print(a)
```

このプログラムでは 1 行目の変数 `a` への代入は、2 行目で同じ変数が上書きされるため、3 行目の `print(a)` に対して無意味であるということを変数 `a` について追いかけることで初めて分かります。

## 3.3 変数の名前

### 3.3.1 プログラムでは複数文字の変数名も使う

先の例では変数名として「`a`」を用いました。数学では変数には 1 文字のアルファベット（やギリシャ文字）を用いることが多いですが、さまざまなデータを取り扱うプログラミング言語では変数名としてより長い名前が使えます。例えば

```
a
x
x2
root
```

`square_root`  
などです。

### 3.3.2 変数の命名ルール

以下のルールを覚えておいてください。

- 英大文字、英小文字、数字、アンダースコア(\_)を使う。
- 大文字と小文字は区別される。
- 数字を先頭に使ってはいけない。
- Python の文法で使用する予約語(例えば if など、IDLE Editor では予約語は赤字で表示されます)は使えない。

日本語（漢字など）の変数名も利用可能ですが、あまり使われていません。

### 3.3.3 分かりやすい変数名を使う

#### 1) 数学での変数名を例に

適切な名称の使用は思考やコミュニケーションを円滑にします。例えば数学でも一次関数を

$$y = ax + b$$

と書けば、 $y$  は  $x$  の一次関数で傾きが  $a$ 、切片が  $b$  だとすぐに分かります。これは  $x$  や  $y$  を変数に、 $y$  は  $x$  の関数として、そして  $a$  や  $b$  はパラメータとして使うという慣習があるからです。

$$b = xa + y$$

は  $a$  と  $b$  を  $x$  と  $y$  に入れ替えただけの式ですが、なんだか急に分かりにくくなります。

#### 2) Python プログラムでの変数名の付け方

プログラムでも変数の命名はプログラムを分かりやすくする重要なものです。以下のようなことに心がけるとよいでしょう。

- プログラムでの意味を表す命名をしましょう。<sup>1</sup>

---

<sup>1</sup> 「命名する」という行為は日常生活ではありません、子供やペットに名前をつけるぐらいでしょう。しかし、コンピュータ（情報）を利用する場合にはファイルやフォルダ名など、命名するという行為が大変重要になります。プログラミングはそれが最もよく現れる行為の一つだと言えます。「命名する」というスキルが必要だということを意識してみてはいかがでしょうか。

- 1文字などの短い変数名はできるだけ狭い範囲だけで有効な変数として使いましょう。特に l, o, O (小文字のエル、小文字のオー、大文字のオー) などは数字の 1(イチ)や 0(レイ、ゼロ)と紛らわしいため使用しないようにします。
- 大文字ではなく、小文字を使いましょう。大文字は値を変化させない定数を表すことに使われることが多いです。
- 複数語の変数名は単語の間をアンダースコア「\_」で区切ります。たとえば「street\_name」など<sup>1</sup>。
- できれば英語を使う。プログラムは当初の意図とはズれて、想定外に成長し多くの方に使われる場合があります。海外の方に使われる場合もあるので、予め英語で命名しておくといいでしょう。<sup>2</sup>

変数名以外にも、どのようにプログラムを書くと分かりやすいかは重要な点です。Python では PEP8[13] というプログラムを書く際のコーディング規約が提唱されています。

### 演習 3-2 さまざまな変数名を利用する練習。

p2-1.py で示したプログラムについてシェル上で実行で構いませんから、変数名をさまざまに代えて練習してみてください。

- 1行目と2行目の変数名を両方、同一のものに変更しなければならないことに注意してください。
- 複数語をアンダースコアで接続した変数名も試すこと。
- また、先頭に数字を用いた場合、予約語を用いた場合などにどのようなエラーが生じるかも併せて試みてください。

## 3.4 変数への代入と値の評価

以下のプログラムを Python シェルで実行してみてください。

```
a = 1
print(a)
a = a + 1
print(a)
```

<sup>1</sup> このほか、2語目以降の語頭を大文字で表記する方法もよく見かけます。この例では「streetName」となります。

<sup>2</sup> 以前行っていた研究でドイツの先生が書いた FORTRAN のプログラムを C 言語に書き直したことがあります。古い仕様の FORTRAN は変数名の長さが制限されていて、これがドイツ語の単語を短くしたものであつたため、まったく意味が分かりませんでした。その先生のプログラムは丁寧な英語での注釈がついていたので、なんとか目的を果たすことができました。

1行目では変数 `a` に 1 を代入しています。

3行目は右辺と左辺に同じ変数 `a` が現れますので、読み方に注意が必要です。このプログラムについて、Python では

- まず代入演算子(=)の右辺 ( $a + 1$ ) を計算します。

- すでに変数 `a` については 1 が代入されていますので、右辺は「`a` の値を評価した結果」の 1 を使って、 $1 + 1$  となり、2 という計算結果が得られます。

- つぎにこの結果が左辺の変数 `a` に代入（上書き）されます。

変数については「名前のついた箱」というイメージで理解するとよいでしょう。

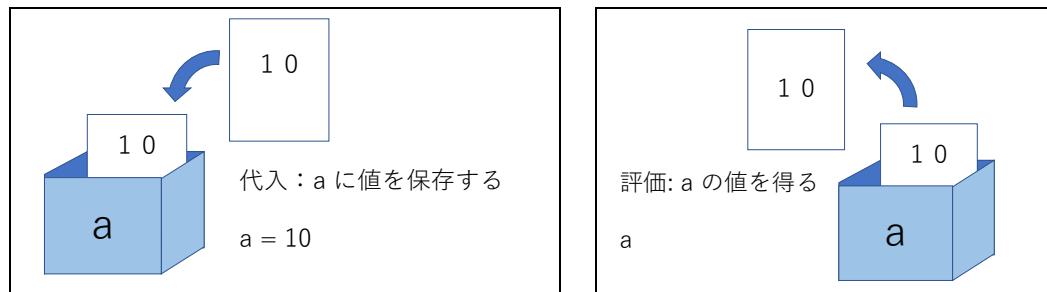


図 3-1 変数についてのイメージ、代入と評価

### 演習 3-3 変数の動作の説明

以下は 1000 円の商品の 15% 引きを計算するプログラムです。

- このプログラムには 1箇所誤りがあり、実行するとエラーになります。どのような誤りがあるかを説明してください。
- 誤りを修正したうえでプログラムの動作を説明してください。

```
kakaku = 1000
nebikiritsu= 15
kakaku = Kakaku*(100-nebikiritsu)/100
print(kakaku)
```

## 3.5 代入演算子

プログラムではある変数に一定数（例えば 1）を加える、一定数を差し引く、という計算をしばしば行います。このような計算を便利に行うため、代入演算子として“=” 以外に以下のようないもも利用可能になっています。

表 3-1 Python の代入演算子

演算子	例	意味
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a*b</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a/b</code>

なお, C 言語などでよく使われる「`++`」や「`--`」という演算子は Python にはありません.

## 3.6 Python で使えるデータ型

先の例ではデータとして整数を扱いましたが, このほか基本的なデータの型として Python では下の表のように小数点以下の値も扱える浮動小数点数, 文字列, 論理値(真と偽)などがあります. Python の特徴として, 整数の桁数に制限を設けていない(コンピュータのメモリ容量や計算速度の問題はありますが,)ということがあります. 例えば

`2**200`

は

`1606938044258990275541962092341162602522202993782792835301376`

と計算してくれます.

数値型としてこのほか, 複素数も扱えます.

表 3-2 Python で使えるデータ型

型 変換のための関数	説明	定数（リテラル） の表記例	備考
整数 <code>int()</code>		12345	Python では桁数に制限はありません
浮動小数点数 <code>float()</code>	小数点以下 を含む数	1.0 2.99792458E8	大きさ(有効数字の桁数と表現できる範 囲)に制限があります E8. という表記は $\times 10^8$ という意味です。
文字列 <code>str()</code>	文字の並び	'aaa' "日本語"	シングルクオートかダブルクオートで文 字列を囲みます
論理値 <code>bool()</code> <sup>1</sup>	条件判断に 使います	True False	定数は先頭が大文字です

なお、Python を含む多くのプログラミング言語で、浮動小数点数の演算は 2 進法で行なわれます。我々は小数も 10 進法で扱っているのですが、 $1/3$  は 10 進法の少数では正しく表現できないのと同じ現象として  $1/10$  が 2 進法の浮動小数点数では正しく表現できません。詳しくはコラム「Float って？」を参照ください。

#### 演習 3-4 データの型の確認

Python シェルで以下を実行してください。

```
a = 1
b = 1/2
c = "ABC"
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

Python ではデータ（一般にオブジェクトと呼びます）には、「型」がありますが、変数にはどのような型のデータでも代入することが可能です。  
変数に代入されているオブジェクトの型を知るには `type()` 関数を使います。

<sup>1</sup> 論理演算の代数としての理論を考案した George Boole にちなんでいます。

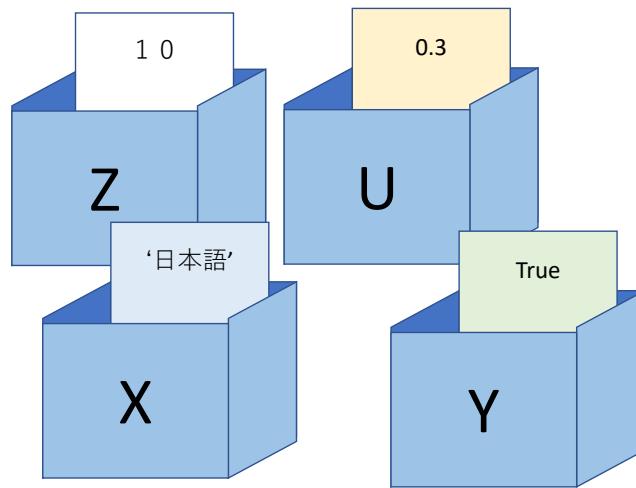


図 3-2 Python では変数の内容の型は自由です

### 3.7 Python の変数のより正しい理解

実際には Python では変数が直接、データ（オブジェクト）を持っているのではなく、「オブジェクトがどこにあるか」という所在の情報（参照）を持っています。今の時点でのことをあまり意識する必要はありませんが、後により複雑なリストなどのデータの扱いを正しく理解する上では重要になります。

変数がもつデータの所在を特定する情報は `id()` という関数で調べることができます。

```
a = 1
b = 2
print(id(a), id(b))
```

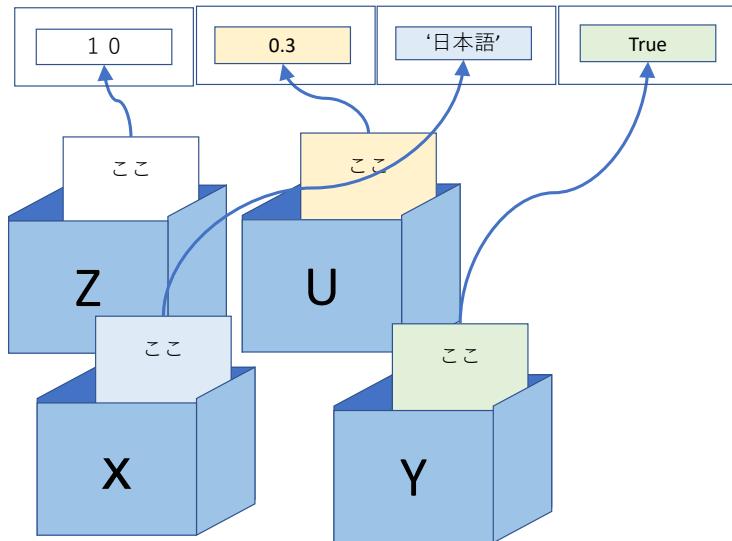


図 3-3 Python の変数はデータ（オブジェクト）の所在情報を持つ

## 3.8 例題：平方根を求める

順次実行と変数の扱いを使った例題として与えられた数値の平方根の近似値を求めてみましょう。以下の計算方法は単純ですが、桁数の多い割り算を使います。このため手で計算するのは面倒ですがコンピュータを使うと簡単に実現できます。

### 3.8.1 計算手順

ある数（例えば 2）の平方根  $\sqrt{2}$  を求める方法として以下が知られています。

- 平方根の近似値を  $r$  とします。分からなければ平方根を求めたい数（この例では 2）と同じ値や 1 としてもかまいません。ここでは  $r = 2$  としましょう。
- 平方根のもう一つの近似値として「平方根を求めたい数」を近似値  $r$  で割った数が考えられます。ここでは  $2/r = 2/2 = 1$  となります。実際に近似値  $r$  が求めたい数の平方根なら、 $2/r$  も平方根になるはずです。
- 新しい近似値  $r^{\text{NEW}}$  として、これら 2 つの中間の値（平均）を考えます。この例では

$$r^{\text{NEW}} = \frac{r + \frac{2}{r}}{2} = \frac{2 + \frac{2}{2}}{2} = \frac{2 + 1}{2} = 1.5$$

となります。

- 今度は  $r$  の値を  $r^{\text{NEW}}$  にして同じことを繰り返します。

$$r^{\text{NEW}} = \frac{1.5 + \frac{2}{1.5}}{2} = \frac{1.5 + 1.33333}{2} = 1.41666$$

皆さんのが知っている 2 の平方根に近づいてきたでしょう。以下、同じ手順を繰り返せばより精度の高い値が得られます。

これをプログラムにするために変数を割り付けて書くと以下のようになります。

- 平方根を求めたい数値 ( $> 0$ ) を変数  $x$  に代入します。
- 平方根の近似値の初期値 ( $> 0$ ) を定め、変数  $r_{\text{new}}$  に設定します。ここでは初期値として  $x$  と同じ値を設定します。
- 変数  $r_1$  に  $r_{\text{new}}$  の値を代入します。
- もう一つの近似値として  $x/r_1$  を考え、変数  $r_2$  に  $r_2 = x/r_1$  として代入します。

$r_1$  が  $x$  の平方根であれば  $r_2$  もまた  $x$  の平方根になりますが、異なっていれば、どちらかが真の値より大きく、どちらかは真の値よりは小さくなり、真の値は  $r_1$  と  $r_2$  の間にあります。

- そこで新しい近似値として  $r_1$  と  $r_2$  の平均  $(r_1 + r_2)/2$  を考え、これで変数  $r_{\text{new}}$  の値を  $r_{\text{new}} = (r_1 + r_2)/2$  と更新します。
- ステップ 3. ~ 5. を適当な回数繰り返します。

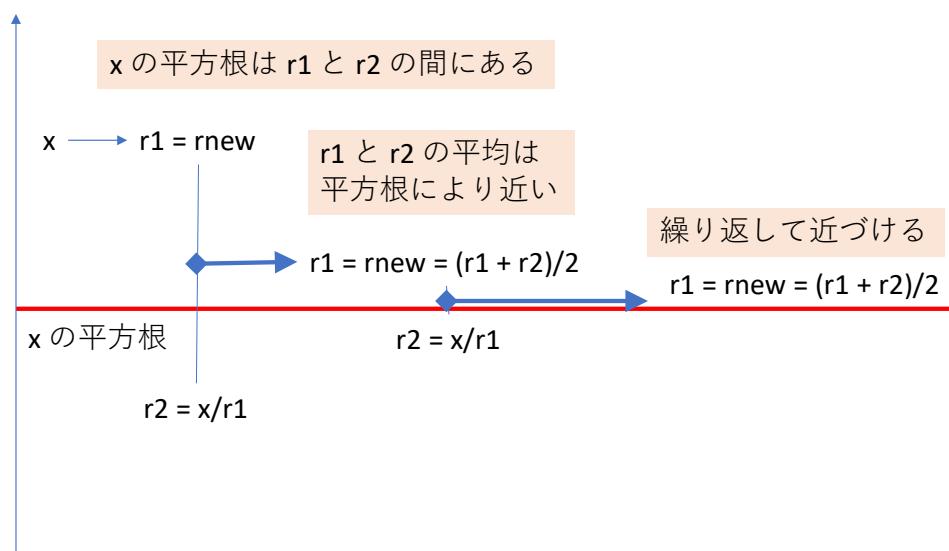


図 3-4 平方根の近似計算の直感的説明

### 3.8.2 Python プログラム

#### 演習 3-5 平方根を求めるプログラムの作成と実行

次の表のソースコードの部分を IDLE エディタで入力し, p3-1.py という名で保存して実行してみてください. (「の」かわりに空白を入力すること)

プログラム 3-1 平方根を求めるプログラム（その 1, p3-1.py）

行	ソースコード	説明
1	# x の平方根を求める	# で始まる部分は注釈
2	x=2	
3	#	
4	rnew=x	最初の近似値の想定
5	#	
6	r1=rnew	
7	r2=x/r1	
8	rnew=(r1+r2)/2	
9	print(r1,rnew,r2)	
10	#	以下は赤字の部分を 3 回繰り返しているだけ.
11	r1=rnew	
12	r2=x/r1	
13	rnew=(r1+r2)/2	
14	print(r1,rnew,r2)	
15	#	
16	r1=rnew	
17	r2=x/r1	
18	rnew=(r1+r2)/2	
19	print(r1,rnew,r2)	
20	#	
21	r1=rnew	
22	r2=x/r1	
23	rnew=(r1+r2)/2	
24	print(r1,rnew,r2)	

以下のような結果が得られれば成功です。

```
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
```

4 回の繰り返しで近似値として 1.4142135623746899 が得られました,  
別途,

**2\*\*(1/2)**

を求めてみると

**1.4142135623730951**

が得られます。小数点以下 11 桁目まで正しいことが分かります。

上では計算方法について直感的な説明を行いましたが、これはニュートン法と呼ばれる数値計算手法の平方根を求める場合への適用例です。詳しくはコラムの「ニュートン法」を参照ください。

### 演習 3-6 エラーを体験する(1).

プログラム 3-1 で 4 行目の `rnew` を誤って `rmew` と綴ってみて実行し、どのようなエラーになるかを確かめてください。17 章「IDLE/Python でのエラーメッセージの読み方」も併せて参考すること。

### 演習 3-7 他の数値の平方根を求める。

1. `p3-1.py` を変更して、他の正の数値の平方根を求めてください。
2. また、このプログラムで 0 の平方根を求めようとすると何が生じるか確認してください。単にエラーのメッセージを見るだけでなく、実際にプログラムをご自身で追いかけて（トレースすると言います）、どこで問題が生じるかを考えてください。

## 3.9 割り算に注意

プログラムでは先の例 `r2 = x/r1` のように変数を「割る数」とする割り算がしばしば登場します、四則演算の中で割り算は「割る数」を 0 とすることができます。 プログラミングで割り算が出てきたら、「割る数」が 0 にならないか、ということを常に気にするようにしましょう。

## 3.10 読み易い式の表記

プログラム `p3-1.py` の 8 行目

```
rnew = (r1 + r2)/2
```

では、代入演算子 `=`、足し算 `+`、割り算 `/`、優先順位を変える `()` が使われていますが、

- 式を読みやすくするため `=` と `+` の前後には空白を入れています。

- 他方で () の内側や / の前後には空白を入れていません。試しにすべての空白を抜くと  
`rnew=(r1+r2)/2`  
 とかなり詰まった感じになり、読みにくくなります。
- 優先順位の高い演算である \* や / の前後はつめ、低い演算である + や -, = の前後を開けるようにするとよいでしょう。

## 3.11 複数の変数への代入

Python では複数の変数への代入を右辺、左辺とも「,」で式や変数を並べることで一つの代入文で実行可能です<sup>1</sup>。

```
a = 1
b = 2
c, d = a*2, b*c
print(c, d)
```

とすれば

2 4

という結果が得られます。

2つの変数の値を入れ替えるには、多くのプログラミング言語では一時的に値を退避させる変数（下の例では tmp）を用いて以下のように書きますが

```
a = 1
b = 2
tmp = a
a = b
b = tmp
```

Python では次のような書き方で可能です。

```
a = 1
b = 2
a, b = b, a
```

## 参考文献

[13] PEP 8 -- Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008/>

---

<sup>1</sup> ここでは説明しませんがタプル(tuple)というデータ形式を背後で用いています。

(2020/2/12 アクセス)

## 4. リスト

---

### 4.1 本章の学習の目標

ここまででは主にデータとしては、 単一の数値や单一の文字列を扱ってきました。 Python では複数のデータを一括して扱う方法がいくつかありますが「リスト」はその代表です。 本章では Python におけるリストの扱いについて以下のことを学びます。

1. リストとはどのようなものかを知る
2. リストの生成法を知る
3. リストの要素へのアクセス方法を知る
4. リストの代入とコピーについて知る
5. リストの他にデータをまとめて扱う仕掛けとしてタプルと辞書の基礎を知る。

### 4.2 Python Shell を用いた学習

本章の内容は短いコードが多く、 エディタで編集して実行するよりは Python Shell で入力しながら動作を確認してゆくことで学習を効率的に進めることができます。 ただし、 以下に注意してください。

- Python Shell は 1 行ずつ処理しますので、 複数行をコピー & ペーストで実行することはできません。 1 行ずつ入力してください。

また、 紙数の節約のため赤字の入力と青字の出力を続けて記載している箇所があります。 Python Shell のプロンプトなどは省略していますが、 赤字の部分を入力し、 青字の表記と実際の出力を確認するという形で学習を進めてください。

### 4.3 リストとは

日常生活では「買い物リスト」といえば、 複数の買うべきものを書き出したメモを指します。 これと同じように、 複数のデータを一括して扱う Python の仕掛けがリスト (list) です。 複数のデータに順番をつけた上で一つのものとして扱えるようになります。 例えば

`a = [5, 1, 3, 4]`

と入力し

```
print(a)
```

とすればリスト全体を

```
[5, 1, 3, 4]
```

を

```
print(a[0])
```

とすれば 0 番目の要素（先頭は 1 ではなく 0 です。）

```
5
```

を

```
print(a[2])
```

とすれば 3 番目の要素

```
3
```

が表示されます。

## 4.4 リストの生成

### 4.4.1 要素を指定した生成

リストの生成は[] 内に「,」で区切って具体的に要素を書く形で

```
a = [5, 1, 3, 4]
```

とか

```
b = ['三条', '四条', '五条', '七条']
```

のように書きます。文字列を要素としてもかまいません。また

```
c = 5
```

```
a = [c, 1, 3, 4]
```

のように変数や式を含んでもかまいません。

同じ要素を持つリストは整数との乗算を用いて以下のように書くこともできます。ただし、この方法は要素がリストなど数値や文字列でない場合は同じオブジェクトを指すように動作するので注意が必要です。

```
a = [1]*4
```

```
a
```

```
[1, 1, 1, 1]
```

### 4.4.2 range() 関数との組みあわせ

空のリストは list クラスのオブジェクトとして

```
e = list()
```

でも生成できます。また、通し番号を生成する `range()` 関数と組み合わせて

```
n = list(range(5))
```

とすると

```
print(n)
```

に対して、0 から 5 未満の数値を要素とするリスト

```
[0, 1, 2, 3, 4]
```

が得られていることが分かります。

#### 4.4.3 文字列からの生成

`range()` の代わりに文字列からリストを生成することも可能です。

```
s = list('abcde')
```

と入力し

```
print(s)
```

とすると

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

が得られ、`s` は文字列 'abcde' を 1 文字ずつ分解したものであることが分かります。

文字列クラスには特定の文字で文字列を区切った単語リストを作る `split()` メソッドがあります<sup>1</sup>。例えば

```
t = "a textbook of Python"
```

```
tlist = t.split()
```

とすると

```
print(tlist)
```

により

```
['a', 'textbook', 'of', 'Python']
```

が得られ、空白を区切りとした単語のリストが得られていることが分かります。

### 4.5 メソッド

先の例で文字列を空白で分割する `split()` というメソッドを用いましたが、Python で扱うデータなどは一般に「オブジェクト」と呼ばれます、その種別ごとにデー

---

<sup>1</sup> このメソッドは引数なしで呼ぶと空白を区切り文字に文字列を分割します。区切り文字の指定も可能ですが。英文ですが例えば Python Shell で `help(str.split)` と入力して解説を読むことができます。

タの型やデータの値に対する操作が予め定められています。このような操作を「メソッド」と呼び、変数名のあとに「.」(ピリオド) を付け、その後にメソッド名と「()」を書いて呼び出します。先の `split()` を利用する例では変数名 `t` の後にピリオドと `split()` を添えて呼び出しています。

```
t = "a textbook of Python"
tlist = t.split()
```

文字列定数にそのままメソッド呼び出しを繋いでもかまいません。上の例なら

```
tlist = "a textbook of Python".split()
```

と書くこともできます。

以下、リストについてもいくつかメソッドを紹介します。ここでは「メソッド」という呼び方と表記法を知っておいてください。<sup>1</sup>

## 4.6 リストの要素へのアクセス

リストの要素へのアクセスは `[]` 内に要素の番号を入れることで行います。

```
a = [5, 1, 3, 4]
print(a[0])
```

により

`5`

が、また

```
a[1] = 2
print(a)
```

により `a` の 2 番目の要素（添え字 1）に `2` が代入され

`[5, 2, 3, 4]`

が得られます。

リストの長さは `len()` 関数で得ます。

```
print(len(a))
```

により

`4`

が得られます。メソッド `a.len()` ではないことに注意してください。

<sup>1</sup> 後の章で紹介する「関数」とメソッドは良く似ていますが、関数は特定のオブジェクトに紐づけられていないのに対し、メソッドは紐づけられたオブジェクトを主な操作の対象とします。呼び出し方はほぼ同じです。メソッドの定義を含むデータの型をプログラマが設定する方法として「クラス」があり、これも後の章で紹介します。

## 4.7 負の添え字とスライス

Python ではリストの添え字の多様な記述が許されています。

### 4.7.1 負の添え字

添え字が負の場合は後ろから添え字の絶対値だけ数えた要素を指します。

```
a = [5, 1, 3, 4]
print(a[-1])
```

に対して、最後の

4

が得られます。下の表参照。

	a[ 5,	1,	3,	4 ]	
正の添え字でのアクセス	a[0]	a[1]	a[2]	a[3]	
負の添え字でのアクセス	a[-4]	a[-3]	a[-2]	a[-1]	

### 4.7.2 スライス

添え字として「先頭番号:終了番号」を与えると、リストの一部を取り出すことができます。これをリストのスライスと言います。終了番号より手前までが含まれることに留意してください。

```
a = [5, 1, 3, 4]
b = a[1:3]
print(b)
```

により

[1, 3]

が得られます。

## 4.8 リストへの追加、結合

リストにはさまざまなメソッドが用意されています<sup>1</sup>。ここではその中でリストに要素を追加する `append()` とリストどうしを結合する `extend()` を紹介します。

---

<sup>1</sup> Python Shell で `help(list)` と入力すると `list` が持つメソッドの説明が読みます。

## 4.8.1 **append** メソッド

リストの最後に引数で与えられた要素を追加します。

```
a = [5, 1, 3, 4]
a.append(2)
print(a)
```

これによりリスト a の最後に 2 が追加され

[5, 1, 3, 4, 2]

が表示されます。

## 4.8.2 **extend** メソッド

2つのリストを統合するには extend() メソッドを使います。

```
a = [5, 1, 3, 4]
b = [2, 6]
a.extend(b)
print(a)
```

により、リスト a の後ろにリスト b の内容が追加され、以下が表示されます。

[5, 1, 3, 4, 2, 6]

注意：append メソッドを使うと以下の例のようにリスト b そのものがリスト a の最後の要素として追加されてしまいます。

```
a = [5, 1, 3, 4]
b = [2, 6]
a.append(b)
print(a)
```

リスト a の後ろにリスト b そのものが追加され、以下が表示されます。

[5, 1, 3, 4, [2, 6]]

## 4.9 リストの代入と複製

まず以下のプログラムを実行してみましょう。

```
a = [1, 2, 3]
b = a
print(a)
print(b)
```

次のような結果が得られるはずです。

[1, 2, 3]

[1, 2, 3]

それでは続けて以下のようなプログラムを実行したら結果はどのようになるか、予想してください。

b[0] = 0

a[1] = 0

print(a)

print(b)

結果は

[1, 0, 3]

[0, 2, 3]

とはならず

[0, 0, 3]

[0, 0, 3]

となります。これは変数 `a` も `b` も全く同一のリストを指しているからで、実際

print(id(a), id(b))

とすると（動作状況により値は異なりますが）、`a` も `b` も同じ `id` を持っていることが分かります。

Python では変数はデータそのものを持つのではなく、「データの所在」を持っています。したがって

b = a

で `b` に代入されるものは `a` の表すデータそのものではなく、`a` の表すデータ（リスト）の所在になります。このため、`b` や `a` の要素への代入は同じリストを操作してしまうのです。

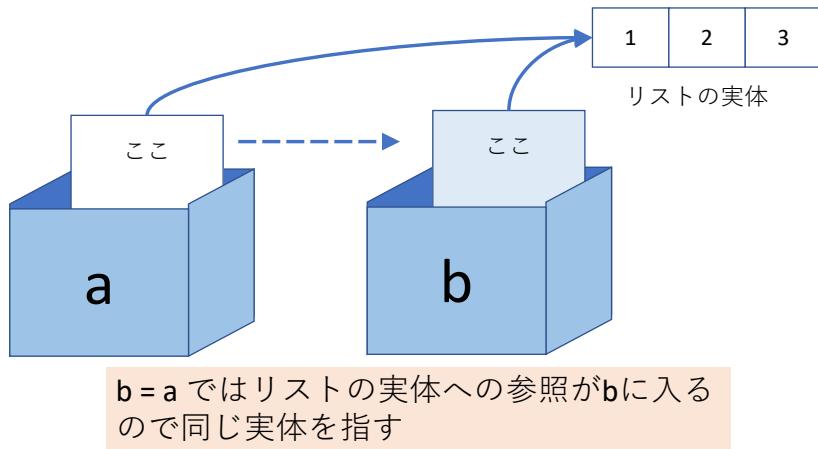


図 4-1 リストの代入

もし、**b** を **a** と独立に操作したいなら、リストの場合は明示的にコピーを作成して代入する必要があります。

**b = a.copy()**

同様の問題は関数の引数にリストを渡した場合にも生じます。コラム「参照と複製」も参考にしてください。

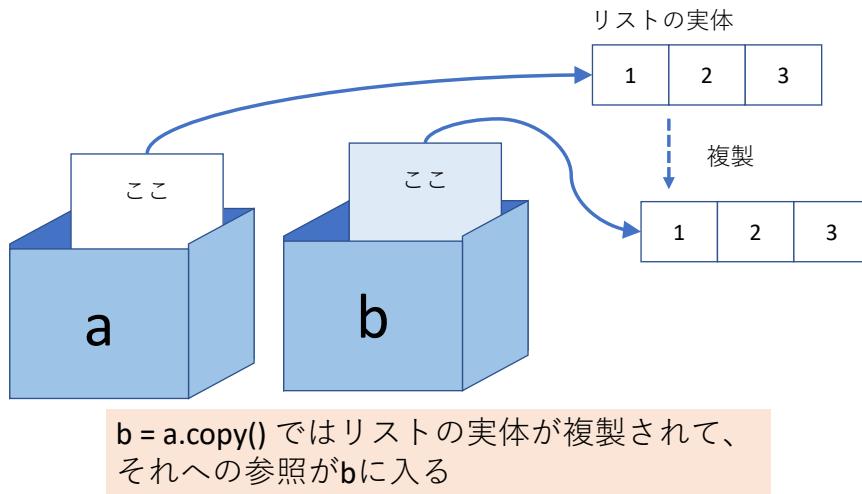


図 4-2 リストのコピー作成と代入

## 4.10 イミュータブルとミュータブル

次のようなコードではプログラムの動作は皆さんの予測と違和感がないはずです。

```
a = 1
b = a
```

```
b = 2
print(a, b)
```

Python のデータの扱いの重要な概念として「イミュータブル」と「ミュータブル」があります。

### 4.10.1 数値や文字列はイミュータブル（変更不能）なオブジェクト

Python では数値や文字列は値を変更することが不可能な「イミュータブル」なオブジェクトとして扱われます。このため、上のプログラムの 3 行目の `b = 2` は `b` が指しているデータ（値は 1）を書き換えるのではなく、`2` というデータを別途作成して、その所在を `b` に代入するのです。実際、値の所在を以下のプログラムで見てみると

```
a = 1
b = a
print(id(a), id(b))
b = 2
print(id(a), id(b))
```

を実行すると

```
>>> a = 1
>>> b = a
>>> print(id(a), id(b))
1434938848 1434938848
>>> b = 2
>>> print(id(a), id(b))
1434938848 1434938880
```

となり、三行目では `a` と `b` は同じ場所を指していますが、5 行目では異なる場所を指しています。

### 4.10.2 リストはミュータブルなオブジェクト

これに対してリストはその要素などの変更を許す「ミュータブル」なオブジェクトとして扱われます。このため、同じリストへの参照をもつ 2 つの変数 (`a, b`) があるとき、`a` や `b` の要素への変更は別の変数の指す内容にも反映されてしまうのです。

## 4.11　浅いコピー、深いコピー

リストに関して、さらに厄介な話で恐縮ですが以下のプログラムで変数 `b` は何を指すでしょうか。

```
a = [[1, 2], [3, 4]]
b = a.copy()
```

試しに以下の操作を続けて行ってみましょう

```
b.append([5, 6])
print(a)
[[1, 2], [3, 4]]
print(b)
[[1, 2], [3, 4], [5, 6]]
```

たしかに `b` は `a` のコピーを代入したので `b` への `append()` は `a` とは独立に行えています。それでは

```
b[0][0] = 0
print(a)
[[0, 2], [3, 4]]
print(b)
[[0, 2], [3, 4], [5, 6]]
```

今度は `b` の`[0][0]` 要素への代入が `a` の要素に反映されています。`a[0]` と `b[0]` が何を指すか調べてみると

```
print(id(a[0]), id(b[0]))
```

に対して

`3188920520008 3188920520008`

となっており、同じオブジェクトを参照しています。

これは `copy()` メソッドが対象となるリスト（コピー元）とは別のリスト（コピー先）を用意して、コピー元の各要素について、その所在をコピー先に写し取っていったことによります。したがって、要素がリストの場合はそのコピーが作られている訳ではありません。このようなコピーを「浅いコピー（shallow copy）」と呼び、要素までコピーを作成することを「深いコピー（deep copy）」と呼びます。

## 4.12 リストを可視化する

Python Tutor というウェブサイト (<http://www.pythontutor.com>) では短い Python プログラムを入力して、その動作（変数の利用）を可視化してくれます。以下はその例です。リストの動作を確認するのに使うと理解の助けになると思います。

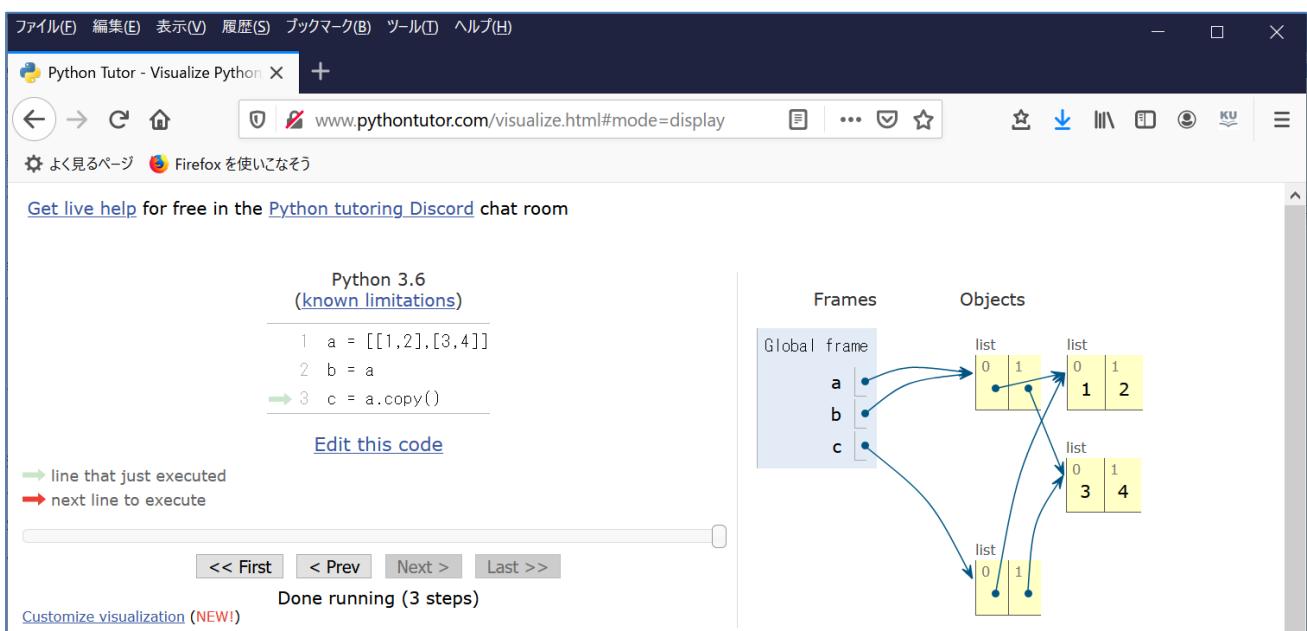


図 4-3 Python Tutor でリストの動作を確認する

## 4.13 計算の過程をリストに残す

前章の平方根の計算 (p3-1.py) について計算結果をリストに残す形に書き換えてみましょう。

プログラム 4-1 リストを用いた計算過程の保存 (p4-1.py)

行	ソースコード	説明
1	# x の平方根を求める	# で始まる部分は注釈
2	x=2	
3	#	
4	rnew=x	最初の近似値の想定
5	rnew_list = [rnew]	rnew_list は rnew を残すリスト
6	#	
7	r1=rnew	
8	r2=x/r1	
9	rnew=(r1+r2)/2	
10	print(r1,rnew,r2)	
11	rnew_list.append(rnew)	更新した rnew を追加

<pre> 12 #  13 r1_=rnew 14 r2_=x/r1 15 rnew_= (r1_+r2)/2 16 print(r1_,rnew_,r2) 17 rnew_list.append(rnew) 18 # 19 r1_=rnew 20 r2_=x/r1 21 rnew_= (r1_+r2)/2 22 print(r1_,rnew_,r2) 23 rnew_list.append(rnew) 24 # 25 r1_=rnew 26 r2_=x/r1 27 rnew_= (r1_+r2)/2 28 print(r1_,rnew_,r2) 29 rnew_list.append(rnew) 30 print(rnew_list) </pre>	<p>以下は同じことを 3 回繰り返しているだけ.</p> <p>最後にリストを出力</p>
--	--

これを実行すると以下の結果が得られます.

```

2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
[2, 1.5, 1.4166666666666665, 1.4142156862745097, 1.4142135623746899]
>>>

```

### 演習 4-1 平方根の計算過程のリストへの保存

プログラム 4-1 を作成し、実行しなさい。また実行後に Python Shell でリスト rnew\_list の大きさと各要素の値を調べなさい。

## 4.14 タプルと辞書

Python にはリストとともにデータをまとめて扱う仕掛けとしてタプルと辞書があります。

### 4.14.1 タプル

タプルはリストと同様に複数の要素からなるデータです。タプルを作るには右辺に単純に値をカンマで区切ってならべて左辺の変数に代入します。

```

a =1,2
a
(1, 2)

```

右辺は丸括弧 () で囲ってもかまいません。

```
a = (1, 2)
```

```
a
```

```
(1, 2)
```

実は複数の値の代入や、複数の値を返す関数からの return 文はタプルとしてデータを扱っていたのです。左辺が要素の個数と同じ数の変数ならそれぞれ要素ごとに代入されます。

```
(b, c) = a
```

```
b
```

```
1
```

```
c
```

```
2
```

タプルの要素はリストと同じように [] に添え字をつけて参照できます。

```
print(a[0])
```

```
1
```

ただし、タプルはイミュータブルなオブジェクトで後から要素に代入はできません。

```
a[0] = 2
```

とすると、以下のようなエラーが生じます。

```
Traceback (most recent call last):
```

```
  File "<pyshell#9>", line 1, in <module>
```

```
    a[0] = 2
```

```
TypeError: 'tuple' object does not support item assignment
```

## 4.14.2 辞書

リストでは添え字として数値で要素にアクセスしました。添え字の並び順に意味があるときには便利な仕掛けです。辞書は数値ではなく「文字列（単語）」で要素にアクセスする仕掛けです。見出しとなる文字列を「キー」と呼びます。

```
age = {"山田":18, "田中":19}
```

```
age
```

```
{'山田': 18, '田中': 19}
```

```
age["山田"]
```

```
18
```

存在しないキーで参照するとエラーになりますが、代入すると追加されます。

`age["佐藤"] = 20`

`age`

`{'山田': 18, '田中': 19, '佐藤': 20}`

キーの有無は `in` 演算で確認します。

`"岡田" in age`

`False`

タプルも辞書もリスト同様に `for` 文の繰り返しの対象として使えます。

### 4.14.3 リスト、タプル、辞書の表記まとめ

まとめたデータを扱うリスト、タプル、辞書では表記のための括弧が異なっています。以下の表に整理しておきます。

表 4-1 リスト、タプル、辞書、表記のまとめ

データの形 式	表記に使う 括弧	例	要素へのア クセス	ミュータブル／ イミュータブル
リスト	[ ]	<code>d = [0, 1, 2]</code>	<code>d[0]</code>	ミュータブル
タプル	( )	<code>d = (0, 1, 2)</code>	<code>d[0]</code>	イミュータブル
辞書	{ }	<code>d = {"a":1, "b":2, "c":3}</code>	<code>d["a"]</code>	ミュータブル



## 5. 制御構造

---

### 5.1 本章の学習の目標

本章ではプログラムの実行を制御する以下の方法を学びます。

1. for 文, while 文による繰り返し処理と range() 関数
2. if 文による分岐
3. try 文によるエラー処理

また、これに関連して以下も学びます

4. 条件式の書き方
5. input() 関数によるキーボードからの入力
6. Python の数学関数
7. 文字出力におけるフォーマット指定

この章で学ぶことはずいぶん多いですが、後の章で実際のプログラムを書く中で何度も登場しますから、その中で自然と身につくと思います。

今の段階で細かなことをすべて覚えるのではなく、「何ができるのか」を知ることと、実際に書いてみることを体験する、と割り切って学習してください。何が書かれていたか覚えてさえいれば、細かな事項は必要になったときに教科書を見返して確認すればすみます。

### 5.2 for 文による繰り返し処理

プログラミングの目的の一つは人が手作業で行うことが難しいぐらいの作業を繰り返すことの自動化です。for 文はこのような繰り返しをプログラミングする重要な方法です。C 言語などに慣れた人にとって Python の for 文は少し構文が異なりますが、リストの要素を対象とした繰り返しを簡単に書けるなどの点で使い勝手がよくなっています。

#### 5.2.1 for 文と range() 関数を用いた一定回数の繰り返し

プログラミングによりコンピュータが威力を発揮するのは多くの処理を速い速度で実施できるときです。ただし、前章で学んだ順次実行では実行するステップ数だけプログラムを書かなければなりません。前章での平方根を求めるプログラムでは

全く同じ記述が 4 回繰り返されています。ここでは、`for` 文と `range()` 関数を用いて繰り返しを自動化することを学びます。

### 演習 5-1 平方根を求めるプログラムへの `for` 文の適用

プログラム 5-1 を作成し、実行してください。

#### プログラム 5-1 平方根を求めるプログラム（その 2, p5-1.py）

行	ソースコード	説明
1	<code># x の平方根を求める</code>	<code>#</code> で始まる部分は注釈
2	<code>x = 2</code>	
3	<code>#</code>	
4	<code>rnew = x</code>	
5	<code>#</code>	
6	<code>for i in range(10):</code>	<code>i</code> を 0 から 9 まで変えながら以下を繰り返します。6 行目最後はコロン「:」があることに注意。
7	<code>    r1 = rnew</code>	Python では繰り返す範囲（ブロック）を字下げ（推奨は空白 4 文字）します。
8	<code>    r2 = x / r1</code>	
9	<code>    rnew = (r1 + r2) / 2</code>	
10	<code>    print(r1, rnew, r2)</code>	

IDLE エディタでは領域を選んで、`Ctrl` キーを押しながら `]` キーを押す(`Ctrl-]`と表記します)一括して字下げできます。逆の操作は `Ctrl-]` です。

### 5.2.2 `for` 文の書き方<sup>1</sup>

「`for`」という英単語にはいろいろな意味がありますが、ここでは「～のために」と考えると分からなくなります。「～について」ぐらいの意味で読み取って下さい。Python の `for` 文は以下のように構成します。

**for 目標変数 `in` 繰り返しの範囲 :**

繰り返すブロック

上の `p5-1.py` の例では「目標変数」は `i` で、「繰り返しの範囲」は `range(10)`、ブロックは（4 文字）字下げされた 7~10 行です。

`range(10)` という関数は 0 から「添え字の値 - 1」の「9」までの 10 個の値を生成する関数で、`for` 文は生成された値を変数 `i` に入れて、ブロックの部分を繰り返します。

この `for` 文を「日本語に訳読する」なら以下のようになるでしょう。

<sup>1</sup> Python の `for` 文は後述のリストなどと組み合わせて効果的な書き方が可能ですが、ここでは `range()` 関数との組み合わせを紹介します。

「繰り返しの範囲」の各値をいれた「目標変数」について「繰り返すブロック」を繰り返す。

以下、Python の文法的事項の説明は上の例のように枠で囲って示します。固定的な表現は赤字で、内容によって変化するものは黒字で書きます。

### 演習 5-2 ブロックの確認

先の例(p5-1.py) の 10 行目をプログラム 5-2 ように左につめてブロックから外し、動作を確認し説明してください。

プログラム 5-2 平方根を求めるプログラム（その 2, p5-2.py）

行	ソースコード	説明
1	# x の平方根を求める	# で始まる部分は注釈
2	x = 2	
3	#	
4	rnew = x	
5	#	
6	for i in range(10):	
7	r1 = rnew	
8	r2 = x / r1	
9	rnew = (r1 + r2) / 2	
10	print(r1, rnew, r2)	この行をブロックから外す。

### 演習 5-3 イタズラ

上のプログラム(p5-2.py)は端末への出力を for 文の繰り返しから外したので、繰り返し部分は高速に実行できます。6 行目の range() 関数の添え字を 10 から 100, 1000, 10000, 100000, 1000000, 10000000 と変化させてどの程度の時間がかかるか試してみてください。<sup>1</sup>

### 5.2.3 Python でのブロック

複数行のプログラムを一括して扱うブロックはプログラミングでの重要な考え方です。

- Python ではブロックを「同じ深さの字下げ」で表記します。これは Python の特徴の一つです。
- ブロックを要求する for などの文の行末はコロン (:) で終わります。

<sup>1</sup> 現代のパーソナルコンピュータは GHz 程度のクロックで動いており、1 命令の実行を 1 クロックで行うなどとても高速な処理を行います。しかしながら、中間コード方式で実行される Python はプログラミング言語の中では処理が遅いものです。

- Python のプログラムでは字下げが重要な意味をもつため、空白 4 文字など統一した記法が望されます。
- IDLE Editor では for 文などブロックを伴う行を入力すると続く行を自動的に字下げしてくれます。
- **字下げに全角の空白を入れてしまうとエラーになります。見た目に分かりにくいくので注意してください。** また TAB キーはエディタの設定にもよりますが、TAB コードがそのまま入力される場合もエラーになります。
- 他の言語では例えば C ではブロックは {} で囲みます。他の言語に慣れた人は記法に注意が必要です。

#### 演習 5-4 エラーを体験する(2).

for 文など後続のブロックを要求する文では最後に「: (コロン)」を書くことが求められます。これを忘れたり、あるいは誤って「; (セミコロン)」を入力すると、どのようなエラーになるかを確かめてください。17 章「IDLE/Python でのエラーメッセージの読み方」も併せて参照すること。

#### 演習 5-5 エラーを体験する(3).

プログラム 5-1 で 7~10 行目は同じ字下げ（空白 4 文字）として同一のブロックとしています。誤って、7 行目の空白を少なくしたり（例えば空白 3 文字）、逆に多くしたり（空白 5 文字）にしたりすると、どのようなエラーになるかを確かめてください。17 章「IDLE/Python でのエラーメッセージの読み方」も併せて参照すること。

### 5.2.4 for 文内の処理の制御

for 文内の処理を打ち切ったり、特定の繰り返しでは処理をスキップしたりするために break と continue という命令が用意されています。

- **break:** for 文の繰り返しから脱出します。
- **continue:** for 文の繰り返しのブロックの残りの部分をスキップして次の繰り返しに移ります。

これらは、後述の if 文での条件分岐と組み合わせて用いられます。

#### プログラム 5-3 continue と break (p5-3.py)

行	ソースコード	備考
1	for i in range(10):	
2	if i==1:	i が 1 なら次に移る
3	continue	
4	if i==8:	i が 8 なら繰り返しから脱出

5	<code>break</code>	
6	<code>print(i)</code>	

このプログラムの実行結果は以下のようになります。

```
0
2
3
4
5
6
7
```

### 演習 5-6 `continue` と `break` の説明

プログラム 5-3 について上記の実行結果をソースコード用いて説明しなさい。

## 5.2.5 `range()` 関数

正確には `range` は関数ではなく「クラス」として実装されているのですが、関数のような使い方が主ですので、ここでは簡単のため `range()` 関数と呼びます。

`range()` は一定間隔での数値の並びを生成してくれます。ただし、実際にどのような値が生成されるかは `range()` 関数の呼び出しだけでは分からないので、生成される値からリストを生成して確認します。Python シェルで

```
list(range(10))
```

と入力すると

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

が得られます。

`range()` 関数の使い方としては引数の与え方で以下の 3 通りがあります。

- 終了値を与える。先に見たように

```
range(終了値)
```

といった使い方です。0 で始まり、終了値より手前の整数で終わります。

プログラミング言語になじみのない方は 0 で始まることや終了値が含まれないことを奇異に思われるかもしれません。これについてはコラム「0 始まり」も参照ください。生成される個数は終了値と一致するので実際にはそれほど分かりにくいものではありません。0 で始まり、指定した終わりの値の手前までを扱うという範囲の取り扱いは Python では標準的に行われます。

- 開始値と終了値の 2 つを与える。

```
range(開始値, 終了値)
```

開始値は含まれ、終了値は含まれません。2つの引数の間にはカンマ「,」を入れてください。カンマの後に（半角の）スペースを入れると読みやすくなります。

- 開始値と終了値とステップ幅の3つを与える。

`range(開始値, 終了値, ステップ幅)`

### 演習 5-7 range() 関数

上で述べたように `list()` と組み合わせて `range()` 関数の3通りの使い方を Python シェルで練習してください。

## 5.2.6 合計の計算

先の例では `range` 関数は繰り返しの回数を指定するためだけに使い、目標変数は特に使用しませんでした。ここでは、目標変数の値を利用する例を練習します。

### プログラム 5-4 合計の計算 (p5-4)

行	ソースコード	備考
1	<code>sum=0</code>	合計を 0 に
2	<code>for i in range(10):</code>	<code>i</code> は 0 から 9
3	<code>    sum+=i</code>	<code>sum</code> に <code>i</code> を加える
4	<code>print(sum)</code>	<code>sum</code> を表示

### 演習 5-8 合計の計算

プログラム 5-4 を実行してみてください。また、合計の範囲を変えてみてください。3行目は `sum += i` と書くこともできます。これを試してみてください。

## 5.2.7 for 文の入れ子

行と列のように2方向に広がる表の各要素の生成は `for` 文のブロック中にさらに `for` 文を書くことで実現できます。例えば

### プログラム 5-5 for 文の入れ子 (p5-5.py)

行	ソースコード	説明
1	<code>for i in range(3):</code>	
2	<code>    for j in range(3):</code>	
3	<code>        print(i,j)</code>	

を実行すると以下のように出力されます

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

外側の `for` 文と内側の `for` 文で「目標変数」を変えていていることに注意してください。

### 演習 5-9 `for` 文の入れ子

プログラム 5-5 の 2 行目の `range()` 関数の引数に変数 `i` を使う (`range(i)` とする) とどうなるか、試してみてください。

## 5.2.8 リストを操作する `for` 文

### 1) リストの長さと `range` 関数を組み合わせる方法

リストの要素を `for` 文で順に操作するには `range(len(a))` で要素の番号を生成することで、例えば

```
a = [5, 1, 3, 4]
for i in range(len(a)):
    print(i, a[i])
```

により

```
0 5
1 1
2 3
3 4
```

が得られます。

### 2) リストを `for` 文で直接使う方法

また、要素の値を参照するだけでよいなら以下のような書き方も可能です。

```
a = [5, 1, 3, 4]
for d in a:
    print(a)
```

により

```
5
1
3
4
```

が得られます。この場合、dには要素の中身が与えられますので、dの値を変更してもリストの中身は変わりません。リストの要素を更新したいときには添え字でアクセスする方法を使います。

### 3) enumerate 関数を使う方法

添え字と要素の値と共に使いたい場合は enumerate 関数を使います。以下の例を参考にしてください。

```
a = [5, 1, 3, 4]
for i, d in enumerate(a):
    print(i, d)
0 5
1 1
2 3
3 4
```

#### 演習 5-10 リストの要素の平均値を求める

数値を要素とするリストの平均値は例えば

```
a = [5, 1, 3, 4]
sum = 0
for i in range(len(a)):
    sum += a[i]
average = sum/len(a)
print(average)
```

で求められ、実行結果は以下のように得られます。

3.25

## 演習 5-11 要素の参照方法の変更

リストを直接 for 文で利用する形に上のプログラムを書き換えなさい。

### 5.2.9 for 文を使ったリストの内包表記

値が添え字の 2 乗のリスト

**[0, 1, 4, 9, 16]**

を作ることを考えます。上のように直接、明示的に書いても構いませんし

```
a = []
for i in range(5):
    a.append(i*i)
```

と書いても構いません。このほか Python では for 文をリストの中に書く内包表記と呼ばれる使い方があります。

**a = [i\*i for i in range(5)]**

## 5.3 while 文による繰り返し

### 5.3.1 精度を指定した平方根の計算

平方根の計算を一定の精度を要求して計算するようにしましょう。r1 と r2 は真の値を挟んでいますので、その差の絶対値  $|r1 - r2|$  は計算の精度として捉えることができます。精度を  $10^{-6}$  以下として指定してプログラムを構成します。

このような精度は数値が小数点以下何桁かを指定する絶対精度ですが、科学の計算では、有効数字の桁数を指定するほうが使いやすいという面もあります。これについてはコラムの「相対精度」を参照してください。

#### プログラム 5-6 平方根を計算するプログラム（その 3, p5-6.py）

行	ソースコード	説明
1	# x の平方根を求める	
2	x_=2	
3	#	
4	rnew_=x	
5	#	
6	diff_=rnew_-x/rnew	2つの近似値 (x と 1 (=x/x)) の差をとる
7	if diff_<0:	負なら符号を反転。

8	<code>diff = -diff</code>	
9	<code>while (diff &gt; 1.0E-6):</code>	差が $10^{-6}$ より大きければ繰り返す
10	<code>    r1 = rnew</code>	
11	<code>    r2 = x / r1</code>	
12	<code>    rnew = (r1 + r2) / 2</code>	差の再計算
13	<code>    print(r1, rnew, r2)</code>	
14	<code>    diff = r1 - r2</code>	
15	<code>    if diff &lt; 0:</code>	
16	<code>        diff = -diff</code>	

### 演習 5-12 プログラム 5-6 の作成と実行

プログラム 5-6 を作成し実行してください。

以下のような実行結果が得られるはずです

```
2 1.5 1.0
1.5 1.416666666666665 1.3333333333333333
1.416666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002
```

### 5.3.2 無限ループ型での平方根の計算

無限ループ型で構成した平方根を計算するプログラムをプログラム 5-7 に示します。break 文で脱出するため if 文で計算精度を判定しています。p5-6.py との違いは以下の 2 点です。

- ループを開始する際に終了条件を判定していません。
- while 文の条件は継続を判定するためのものですが、break 文を発動する条件は終了を判定するためのものであり、条件は逆になります。

プログラム 5-7 平方根を求めるプログラム（無限ループ型, p5-7.py）

行	ソースコード	説明
1	<code># x の平方根を求める</code>	
2	<code>x = 2</code>	
3	<code>#</code>	
4	<code>rnew = x</code>	
5	<code>#</code>	
6	<code>while True:</code>	無限ループ型の繰り返し
7	<code>    r1 = rnew</code>	
8	<code>    r2 = x / r1</code>	
9	<code>    rnew = (r1 + r2) / 2</code>	
10	<code>    print(r1, rnew, r2)</code>	
11	<code>    diff = r1 - r2</code>	

12	<code>if diff &lt; 0:</code>	
13	<code>    diff = -diff</code>	
14	<code>if diff &lt;= 1.0E-6:</code>	
15	<code>    break</code>	差の絶対値が $10^{-6}$ 以下なら break で終了

### 5.3.3 while 文の構成

条件が成立している間、ブロック内の処理を繰り返す while 文の書式は以下のようになっています。

**while** 条件式:

実行ブロック

条件は実行ブロックに入る「前」にチェックされます。このため先の例では diff の値を while 文に入る前と実行ブロック内の両方で計算しています。条件式の詳しい書き方についてはこの先で紹介します。

また for 文と同様、繰り返しを脱出する break や次の繰り返しに移る continue が使えます。

### 5.3.4 無限ループ<sup>°</sup>

while 文の使い方として以下のような例に出会うことも少なくありません。

**while True:**

break 文を含む実行ブロック

この例では条件式は真であることを表す定数 True ですので、常に成り立ち、while 文そのものは無限にループする仕組みとなっています。そのため実行ブロック内で脱出のための条件を確認し、break 文で脱出する必要があります。

実際に脱出のための条件が成り立たない場合は Python の実行を強制的に止めなければなりません。キー操作「Ctrl-C」で停止させてください。

## 5.4 if 文による分岐

### 5.4.1 if 文の構成

if 文は条件が成立している場合のみ与えられたブロックを実行するもののほか、成立しない場合 (else) に実行するブロックを指定するもの、成立しない場合にさらに別の条件を検査するもの (elif, else if の意味) など、いくつかのパターンがあります。

**if** 条件式:

成立した場合に実行するブロック

**if** 条件式:

成立した場合に実行するブロック

**else:**

成立しない場合に実行するブロック

**if** 条件式 1:

条件式 1 が成立した場合に実行するブロック

**elif** 条件式 2:

条件式 1 は成立しないが条件式 2 が成立する場合に実行するブロック

**else:**

いずれの条件も成立しない場合に実行するブロック

なお、C 言語の switch 文のように検査する値によって 3つ以上のブロックの実行を切り替える命令はありません。elif を多数回使うことで同様の分岐ができます。

## 5.4.2 条件式の書き方

### 1) 数値の比較

条件の代表的なものは数値の比較です。比較するための演算子として次のものが用意されています。

表 5-1 Python の比較演算子

演算子	意味	備考
<code>==</code>	等しい	等号は 2 つです。これは 1 文字の場合を代入演算に使うためです。
<code>!=</code>	等しくない	2 文字です
<code>&gt;</code>	左辺が右辺より大きい	
<code>&lt;</code>	左辺が右辺より小さい	
<code>&gt;=</code>	左辺が右辺以上で	2 文字です
<code>&lt;=</code>	左辺が右辺以下	2 文字です

「等しい」を判定する演算子は等号 2 つ (`==`) です。間違いやすいので注意しましょう。

また、浮動小数点数(float)は多くの場合、近似値を扱いますので、「等しい」という比較は予期せぬ振る舞いをする可能性があります。この使用を避け、不等号で判断するようにしてください。

## 2) 文字列の比較

文字列についても上記の演算が使えます。ただし、大小関係は文字コード(unicode)としての番号で行われますので使用は慎重に行ってください。

また、上記に加えて「in」で左辺が右辺の文字列に含まれているかどうか調べることができます。例えば

```
'a' in 'abc'
```

に対して

```
True
```

が得られます。

## 3) 論理演算

複数の条件を合成するために論理演算子 「and」, 「or」, 「not」 が用意されています。

## 4) () による演算の優先

Python では演算子に優先順位が定義されており、

- 算術演算は比較演算よりも優先,
- 比較演算は論理演算よりも優先

されます。プログラムを読みやすくするために () で明示的に優先順位を示した方がいいでしょう。例えば

```
a == 1 and b != 0
```

より以下のほうが動作は同じですが、読みやすいでしょう

```
(a == 1) and (b != 0)
```

### 5.4.3 if 文の入れ子

for 文を入れ子にして利用したように if 文を入れ子にして利用することもしばしば行われます。次の 2 つのプログラムは同じ判定を異なる書き方で書いたもので

す。

### プログラム 5-8 複合的な条件を用いた分岐 (p5-8.py)

行	ソースコード	説明
1	a = 1	
2	b = 0	
3	if (a==1) and (b==0):	
4	print("YES a==1 and b==0")	複合的な条件での分岐

### プログラム 5-9 if 文を入れ子にした分岐 (p5-9.py)

行	ソースコード	説明
1	a = 1	
2	b = 0	
3	if a==1:	
4	if b==0:	if 文を入れ子にした分岐
5	print("YES a==1 and b==0")	

### 演習 5-13 エラーを体験する(4).

if 文内で比較演算子 == の代わりに誤って代入演算 = を書いてしまうことはよくある間違いの一つです。プログラム 5-9 プログラム 5-1 の 3 行目を if a = 1: と書いた場合に、どのようなエラーになるかを確かめてください。17 章「IDLE/Python でのエラーメッセージの読み方」も併せて参照すること。

## 5.5 端末からの入力

これまで平方根を計算したい数値をプログラムに埋め込んで計算してきました。端末から入力する方法を考えましょう。以下は Python シェルの画面です。赤字は入力、青字は出力です。

Python Shell の画面	備考
<pre>&gt;&gt;&gt; a = input("*** ") *** sss &gt;&gt;&gt; a 'sss' &gt;&gt;&gt; type(a) &lt;class 'str'&gt; &gt;&gt;&gt;</pre>	<p>“*** ” を入力促進の文字列にして input 関数で入力を得て a に代入 sss と入力 a の値を評価</p> <p>type 関数で a のデータ型を調査 文字列型(str)であると表示</p>

`input` 関数の引数（() 内に書く文字列）は端末に表示する文字列です。また、返り値（呼び出し結果）のデータ型は文字列です。

数値データを得るにはこれを `int()` や `float()` で適した型に変換します。  
先の平方根を求めるプログラムでは `x` の値を設定している箇所を

```
x = input("平方根を求める数 ")
```

```
x = float(x)
```

に入れ替えるか、あるいは一括して

```
x = float(input("平方根を求める数 "))
```

と書きかえることで変数 `x` に端末から入力された数値を得ることができます。

### 演習 5-14 平方根を求める値の端末からの入力

`p5-6.py` を改造して端末から平方根を求める数値を入力するようにしなさい。`p5-6.py` もしくは `p5-7.py` を開き、File メニューの Save As で `ex3_3.py` として保存したうえで改造すること。

## 5.6 エラーへの対処

関数 `float()` や `int()` は引数として与えられた文字列が数値として解釈できない場合 `ValueError` という種類のエラーを発生します。特にその場合の処理を指定しないければ Python はそこで処理を中断します。プログラム内でエラーを処理するには `try` 文を使います。

次のプログラムは継続的に入力を受け取りエラー処理を行い、正の数値の場合のみ `print(x)` で値を出力します。このプログラムは無限ループを構成していて、停止条件を明示的には書いていません。プログラムを停止するためには `Ctrl-C` を入力してください。

### プログラム 5-10 入力を得て検査するプログラム(`inputcheck.py`)

行	ソースコード	説明
1	<code>while True:</code>	無限ループ
2	<code>    x = input("正の数値を入力してください")</code>	
3	<code>    try:</code>	エラーが生じる箇所を <code>try</code> ブロックに入れる
4	<code>        x = float(x)</code>	<code>ValueError</code> への対応
5	<code>    except ValueError:</code>	大文字小文字に注意
6	<code>        print(x, "は数値に変換できません")</code>	
7	<code>        continue</code>	
8	<code>    except:</code>	その他のエラーへの対応
9	<code>        print("予期していないエラーです")</code>	

10	<code>exit()</code>	終了する, <code>try</code> ブロック はここまで,
11	<code>if x&lt;=0:</code>	正の値であることの検査
12	<code>print(x, "は正の数値ではありません")</code>	
13	<code>continue</code>	
14	<code># 以下は正しい入力が得られた時の処理</code>	
15	<code>print(x)</code>	<code>while</code> ブロックのつづき

### 演習 5-15 エラー処理の確認

プログラム 5-10 を実行し、さまざまな入力で動作を確認してください。

### 5.6.1 `try` 文の構成

`try` 文では例外を生じるブロックを `try` 文の中に入れ、`except` 文で例外を指定して処理するブロックを書きます。例外が指定されていない `except` 文は（その上で処理されるものを除いて）すべての例外に対して機能します。

`try:`

例外処理の対象とするブロック

`except 例外:`

その例外が生じた際の処理を行うブロック

`except:`

上記で指定した以外のすべての例外に対応するブロック

### 5.6.2 外部からの入力は疑え

外部から与えられる入力はプログラマーには統制できません。正しい入力だけを想定して書かれたプログラムは想定外の入力に正しく応答できず、場合によっては間違った結果を出してしまう恐れもあります。外部からの入力については「疑ってかかる」ことが重要で、値の妥当性を検査したり、生じえるエラーの処理を的確に行ったりすることが望まれます。

## 5.7 Python での数学関数

これまでの例では誤差 `diff` の絶対値を「負の場合は符号を反転する」という方法で

```
if diff < 0:  
    diff = -diff
```

と明示的に計算していました。Python では絶対値関数 `abs()` が利用可能ですが、上の計算は 1 行で

```
diff = abs(diff)  
を書けます。
```

また Python には数学関数を利用するためのライブラリ（モジュール） `math` が提供されています。これを用いるためには利用に先立って

```
import math
```

という命令でモジュールを導入します。ここで定義されている定数や関数は

```
math.pi  
math.sqrt(2)
```

のようにモジュール名の後に「`.`」で呼び出したい定数や関数を書きます。上の例は円周率と平方根です。

### 演習 5-16 math モジュールの使用

上の例に従って Python Shell で `math` モジュールを使ってみてください。

## 5.8 数値データ・文字列の変換、文字列の結合

`input()` 関数では文字列が得られ、これを数値データに変換するために `int()` や `float()` などの関数を使いました。他方、数値データを文字列に変換するには `str()` 関数や書式を指定する場合には次に述べる `format()` メソッドを使います、ここでこれらを整理しておきましょう。

表 5-2 数値と文字列の相互変換

変換	関数	例	備考
文字列→整数	<code>int()</code>	<code>a = int("123")</code>	不適切な文字列では <code>ValueError</code> というエラーが生じます
文字列→浮動小数点数	<code>float()</code>	<code>a = float("123.4")</code>	同上
整数, 浮動小数点数→文字列	<code>str()</code>	<code>s = str(123.4)</code>	<code>str()</code> は数値データだけでなく、リストなどさまざまなオブジェクトを文字列に変換します。

なお、`print()` 関数に数値データなど文字列以外のデータを渡したときには自動で文字列に変換されます。

文字列の結合は単純に文字列どうしを「`+`」演算子でつなぎます。このほか、「`*`」演算子は整数型の値と組み合わせて文字列を繰り返すことができます。数値

データを文字列と結合するには `str()` 関数でまず文字列に変換してから結合します。

表 5-3 文字列の結合と繰り返し

操作	演算子	例	結果
文字列の結合	+	<code>"abc" + "def"</code>	<code>"abcdef"</code>
文字列と数値の結合		<code>"abc"+str(1.2)</code>	<code>"abc1.2"</code>
文字列の繰り返し	*	<code>"abc"*2</code>	<code>"abcabc"</code>

## 5.9 数値を表示する際のフォーマット指定

Python の `print()` 関数で数値を表示すると、与えられた数値に合った桁数などが自動で選ばれます。表示する桁数をそろえるなど、利用者が表示する書式を指定することも可能です。具体的な例として

```
c = 2.99792458E8
na = 6.02214076E23
form = '光速は{0:12.8g} m/s, アボガドロ数は {1:12.8g} mol**(-1) です'
print(form.format(c, na))
```

といった感じで利用します<sup>1</sup>。実際に実行すると以下の結果を得ます。

光速は 2.9979246e+08 m/s, アボガドロ数は 6.0221408e+23 mol\*\*(-1) です

- 3 行目の右辺は書式を指定する文字列で {} で囲まれた箇所が数値を変換する書式です。
  - 例えば `{0:12.8g}` は後で述べる `format` メソッドの引数の 0 番目(書式のコロンの手前の数字)の要素に対して、
  - 最小 12 桁とて、小数点以下 8 桁まで g 形式（浮動小数点数を表示する形式の一つ）で表示する指示を意味します。
  - 形式には整数を 10 進数で表示する場合は'd'、浮動小数点数を指数表記で表すには 'e'、固定小数点表記で表すには 'f' が用いられます。両者を値によって切り替えるのが 'g' 形式です。
- 4 行目の `form.format(c, na)` は文字列 `form` を書式に変数 `c` と `na` を変換した文字列を生成することを意味します。文字列変数（ここでは `form`）に付随するメソッド（関数のようなもの）`format` を呼び出して実施するのですが、メソッドは対象となる変数に「.」で続けて指定しています。

<sup>1</sup> 国際単位系(SI)では 2018 年の改定で 2019 年 5 月 20 日から「国際キログラム原器」が廃され、単位系が定義として与えられた物理定数で組み立てられるようになりました。ここで例として挙げた光速やアボガドロ数も計測して定めるものではなく、確定された量として定義されています。

## 5.10 力試し

### 演習 5-17 力試し

`inputcheck.py`, `p5-6.py` を組み合わせて以下の条件を満たす平方根を求めるプログラム作成しなさい。

以下の項目に一度に取り組むのではなく、これまでに学んだどの節が関係するかを整理し、1項目づつプログラムを改修しては動作を確認するようにしてください。

1. 絶対値の計算には `abs()` 関数を用いること。
2. 平方根を求める数を繰り返し端末から入力できるようにすること。
3. 平方根を求める数の入力が数値に変換できない場合は、その旨を示して、次の入力を求めること。
4. 平方根を求める数が 0 以下の場合は、その旨を示して、次の入力を求めること。

できれば以下にも挑戦すること

5. 端末からの入力が “end” という文字列なら終了すること。
6. 計算精度を絶対精度ではなく、相対精度で  $10^{-6}$  とすること。これについて大きな数や小さな数（例えば  $10^{10}$  や  $10^{-10}$ ）の平方根を求め、結果を確認すること。

## 6. 京都の交差点を作る

### 6.1 本章の学習の目的

本章ではリストと for 文の組み合わせについて、碁盤目の街路構成を持つ京都の交差点を例に練習します。

### 6.2 京都の交差点を作る

リストを対象とする for 文の例として以下の例を試してみてください。半角と全角文字が混じるので注意してください。地名以外は空白を含めて半角文字であることに注意してください。

プログラム 6-1 (p6-1.py)

行	ソースコード	説明
1	tozai = ["三条", "四条", "五条"]	全角文字は赤字の部分だけ
2	nanboku = ["堀川", "烏丸", "河原町"]	
3	for i in tozai:	
4	for j in nanboku:	
5	cross = i+j	文字列を + 演算で連結
6	print(cross)	

IDLE エディタで編集している画面は以下のようになります。for や in などの予約語、”三条”などの文字列、print 関数はそれぞれ着色されていることを確認してください。

```
program13.py - C:/Users/hjkit/Documents/Python Scripts/textbook/program13.py (...)
File Edit Format Run Options Window Help
1 tozai = ["三条", "四条", "五条"]
2 nanboku = ["堀川", "烏丸", "河原町"]
3 for i in tozai:
4     for j in nanboku:
5         cross = i+j
6         print(cross)
7
Ln: 7 Col: 0
```

結果は以下のようになります<sup>1</sup>

[三条堀川](#)

[三条烏丸](#)

[三条河原町](#)

[四条堀川](#)

[四条烏丸](#)

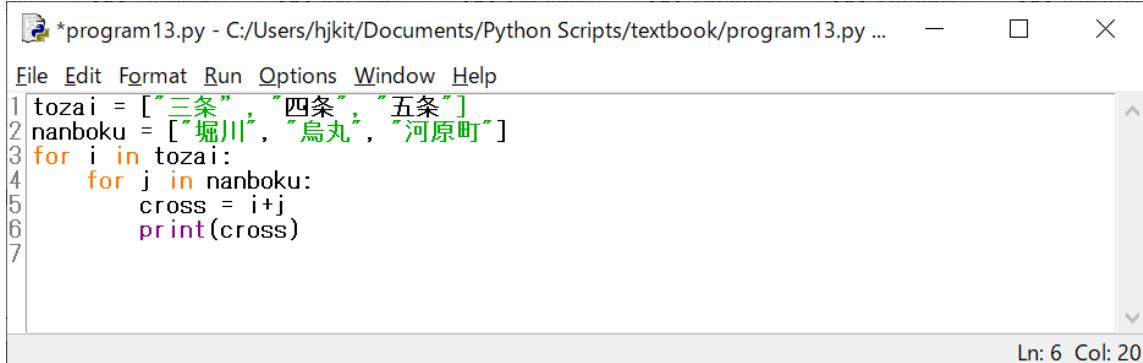
[四条河原町](#)

[五条堀川](#)

[五条烏丸](#)

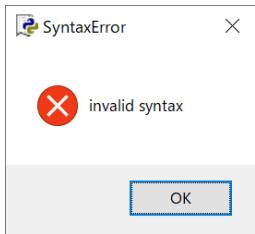
[五条河原町](#)

文字列を閉じる " を誤って全角文字で書いてしまうと着色範囲が変わってしまいます。



```
*program13.py - C:/Users/hjkit/Documents/Python Scripts/textbook/program13.py ...
File Edit Format Run Options Window Help
1 tozai = ["三条", "四条", "五条"]
2 nanboku = ["堀川", "烏丸", "河原町"]
3 for i in tozai:
4     for j in nanboku:
5         cross = i+j
6         print(cross)
7
Ln: 6 Col: 20
```

これを実行するとエラー (invalid syntax, 文法的に正しくない) になります。エディタ上で「四」のところが赤く表示されていますが、"三条", " と"四条"を始める二重引用符が'"三条' 以下の文字列を終わる二重引用符と解釈されるためです。



```
program13.py - C:/Users/hjkit/Documents/Python Scripts/textbook/program13.py ...
File Edit Format Run Options Window Help
1 tozai = ["三条", "四条", "五条"]
2 nanboku = ["堀川", "烏丸", "河原町"]
3 for i in tozai:
4     for j in nanboku:
5         cross = i+j
6         print(cross)
7
Ln: 6 Col: 20
```

<sup>1</sup> 京都の交差点の名称は「四条河原町」のように東西の通りを南北の通りより先に言う場合と「東山三条」のように後に言う場合があります。ここでは機械的に生成しているので、実際の呼称とは一致しません。

## 6.3 リストのリストとその走査

Python のリストの要素は数値や文字列だけでなく、リストでも構いません。これにより、「表のようなデータ」を作成することができます。

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

分かりやすくするため改行を入れて示すと以下のようになります<sup>1</sup>。

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

リスト a 全体を表示

```
print(a)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

リスト a の最初の要素を表示、要素そのものがリストです。

```
print(a[0])
[1, 2, 3]
```

リスト a の最初の要素の 1 番目の要素を表示

```
print(a[0][1])
2
```

表形式のデータを「リスト」の「リスト」として表した場合、すべての要素を参照するには for 文を入れ子にします。いくつか方法があります。

### 6.3.1 添え字を利用する方法

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sum = 0
for i in range(len(a)):
    for j in range(len(a[i])):
        sum += a[i][j]
print(sum)
```

により以下を得ます

45

---

<sup>1</sup> Python の文は通常、行末までですが、Python 3 では上の例のように明らかに継続するものがある（[ に対する ] がないなど）の場合は、改行してもエラーになりません。そうでない場合は行末に「\」が必要です。

### 6.3.2 リストを直接 for 文で扱う方法.

この例ではリストの要素の値を参照するだけですので以下のように書くこともあります。for 文の目標となる変数には番号でなくリストの要素そのものが設定されますので、意味が分かりやすいように row, element を使いました。

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sum = 0
for row in a:
    for element in row:
        sum += element
print(sum)
```

### 6.3.3 京都の交差点の表を作成する

#### 演習 6-1 京都の交差点の表の作成

プログラム 6-1 を参考に以下のようないいな表をリストのリストとして作成してみてください。

```
cross_table = [["三条河原町", "三条烏丸", "三条堀川"],
               ["四条河原町", "四条烏丸", "四条堀川"],
               ["五条河原町", "五条烏丸", "五条堀川"]]
```

プログラムする方策はいくつか考えられます。例えば

- 予めすべての要素が空の文字列 "" であり、必要な要素数を持つリストのリストを作成し、各要素に交差点名を代入する。

#### プログラム 6-2 (p6-2.py)

行	ソースコード	説明
1	tozai = ["三条", "四条", "五条"]	
2	nanboku = ["堀川", "烏丸", "河原町"]	
3	cross_table = [[ "", "", "" ], [ "", "", "" ], [ "", "", "" ]]	
4	for i in range(len(tozai)):	必要な大きさのリストを作る
5	for j in range(len(nanboku)):	
6	cross = tozai[i] + nanboku[j]	交差点名の生成
7	cross_table[i][j] = cross	交差点名の代入

実行後に cross\_table を調べれば表が出来ていることが分かります。

```
>>> cross_table
```

`[['三条堀川', '三条烏丸', '三条河原町'], ['四条堀川', '四条烏丸', '四条河原町'], ['五条堀川', '五条烏丸', '五条河原町']]`

上のプログラムでは表が小さいので 3 行目では直接、リストのリストを定義していますが、大きなものを扱うには以下のようにすればいいでしょう。以下は 5 行、4 列の例。

```
cross_table = []
for i in range(5):
    row = []
    for j in range(4):
        row.append("")
    cross_table.append(row)
```

- `cross_table` として空のリストを設定し、東西の通り上の交差点のリストを作成し、これを `cross_table` に追加 (`append`) してゆく。

#### プログラム 6-3 (p6-3.py)

行	ソースコード	説明
1	<code>tozai = ["三条", "四条", "五条"]</code>	
2	<code>nanboku = ["堀川", "烏丸", "河原町"]</code>	
3	<code>cross_table = []</code>	必要な大きさのリストを作る
4	<code>for i in range(len(tozai)):</code>	通りのリスト
5	<code>    street = []</code>	
6	<code>    for j in range(len(nanboku)):</code>	交差点名の生成
7	<code>        cross = tozai[i] + nanboku[j]</code>	通りのリストについて
8	<code>        street.append(cross)</code>	表に通りのリストを追加
9	<code>    cross_table.append(street)</code>	

### 6.3.4 交差点の表を表示する

#### 1) リストの出力

同じ東西の通り上の交差点名の間に（だけ）は「,」（半角のカンマとスペース）を挿入し、行末にカンマがあってはいけないという形式でリストを出力することを考えてみましょう。具体的には以下のような出力になります。

三条河原町, 三条烏丸, 三条堀川

`print` 関数では通常は改行しますが、`end` オプションを指定すると最後に表示する文字列を変えることができます。

```
street = ["三条河原町", "三条烏丸", "三条堀川"]
```

```

for i in range(len(street)):
    if i < len(street)-1:
        print(street[i], end=", ")
    # 最後以外は ", " を付加
    else:
        print(street[i]) # 最後は改行

```

少し高度ですがリストの要素 "\*" 表現で一括して `print` 関数に渡し、間の区切り文字を `sep` オプションで指定する方法もあります。

```

street = ["三条河原町", "三条烏丸", "三条堀川"]
print(*street, sep=", ")

```

### 演習 6-2 リストのリストの出力

先の例でリストのリストとして作成した交差点の表を以下の条件に沿う形で画面に出力するプログラムを作成してください。

- 1行には、1つの東西の通り上の交差点名を出力する。
- 同じ東西の通り上の交差点名の間には「, 」(半角のカンマとスペース) を挿入する。ただし、行末にカンマがあってはいけない。

具体的には以下のようない出力になります。

```

三条河原町, 三条烏丸, 三条堀川
四条河原町, 四条烏丸, 四条堀川
五条河原町, 五条烏丸, 五条堀川

```

## 7. 関数を使った処理のカプセル化

### 7.1 本章の学習の目標

1. 本章では前章の例題を使って、まとめた処理を関数として定義して使用することを学びます。
2. 関数に値を引数として渡すことを学びます。
3. 関数から呼び出し側に値を返り値として戻すことを学びます。
4. 関数内で使う変数とその影響する範囲について学びます。
5. 関数が利用されるパターンの類型を学びます。

前章と同様、この章でも多くのことを学びますが、後の章で具体的に使用する中で身に着けることができますので、今の段階で細かなことをすべて覚えるのではなく、「何ができるのか」を知ることと、実際に書いてみることを体験する、と割り切って学習してください。何が書かれていたか覚えてさえいれば、細かな事項は必要になったときに教科書を見返して確認すればすみます。

### 7.2 絶対値関数を作ってみる

関数を定義する簡単な例題として前章で使った絶対値関数を自分で構成してみましょう。

プログラム 7-1 絶対値関数の実装例 (p7-1.py)

行	ソースコード	備考
1	def myabs(x):	関数 myabs の定義、引数は x
2	if x<0:	x が負なら値を反転
3	x=-x	x を返す
4	return x	
5		ここからメインプログラム（実行時に最初に動く部分）
6	while True:	
7	a=float(input(">"))	a を引数に myabs を呼び出している
8	print(a, myabs(a))	

プログラム 7-2 絶対値関数の実装例（その 2, 可能なところで `return` する,  
**p-7-2.py**）

行	ソースコード	備考
1	<code>def myabs(x):</code>	関数 <code>myabs</code> の定義, 引数は <code>x</code>
2	<code>    if x&lt;0:</code>	<code>x</code> が負なら
3	<code>        return -x</code>	値を反転して返す
4	<code>    return x</code>	<code>x</code> を返す
5		
6	<code>while True:</code>	ここがメインプログラム（実行時に最初に動く部分）
7	<code>    a=float(input("&gt;"))</code>	<code>a</code> を引数に <code>myabs</code> を呼び出している
8	<code>    print(a, myabs(a))</code>	

### 演習 7-1 絶対値関数を作る

プログラム 7-1, プログラム 7-2 を作成し動作を確認しなさい。

### 演習 7-2 エラーを体験する(5).

プログラム 7-1, プログラム 7-2 は引数が数値であることを想定しています。例えばプログラム 7-2 の 7 行目で `float()` 関数の呼び出しを忘れて `a = input(">")` と書いてしまうと変数 `a` には文字列が代入されます。この場合に、どのようなエラーになるかを確かめてください。17 章「IDLE/Python でのエラーメッセージの読み方」も併せて参考すること。

## 7.3 関数定義の書式

関数定義の書式は以下のようになります。

**def** 関数名(引数):

説明文字列

関数として実行するブロック

**return** 戻り値としてもどす値

- 関数名は変数名と同様なルールで定めます（一般に識別子）と呼びます。関数は動作を伴うものですから、関数名にはしばしば動詞が選ばれます。
- 引数は受け取りたい値を関数内で使う変数名（仮引数）で書きます。複数の引数を受け取るには仮引数を「,」で区切れます<sup>1</sup>。
- 引数がない場合でも `def` 文の関数名のあとに`()`は省略できません。引数のない関数を呼び出すときも `()` は必要です。

<sup>1</sup> Python にはこれ以外の引き数の扱い方もあるのですが、ここでは省略します。

- 説明文字列 (docstring と呼ばれます) は省略してもかまいません。これについてはコラム「プログラムの文章化」も参考にしてください。
- return 文は通常は関数定義の最後に書きます。
- return 文は特定の条件の成立を検査する if 文のブロック内など、関数定義の最後以外の場所に書いてかまいません。
- 返り値が不要な関数なら return 文は要りません。

## 7.4 仮引数と実引数

関数を呼び出す側の引数を実引数、呼び出される側の引数を仮引数と呼びます。コラム「仮引数と実引数」も参照してください。

- 実引数は変数ではなく、式や関数呼び出しでもかまいません。まず式として実引数を評価して、その結果を関数に渡します。
- 実引数と仮引数は同じ名前であることは要求しません。
- 数値や文字列などが仮引数で関数に渡された場合は関数内で仮引数に別の値を代入しても実引数の値には影響しません<sup>1</sup>。

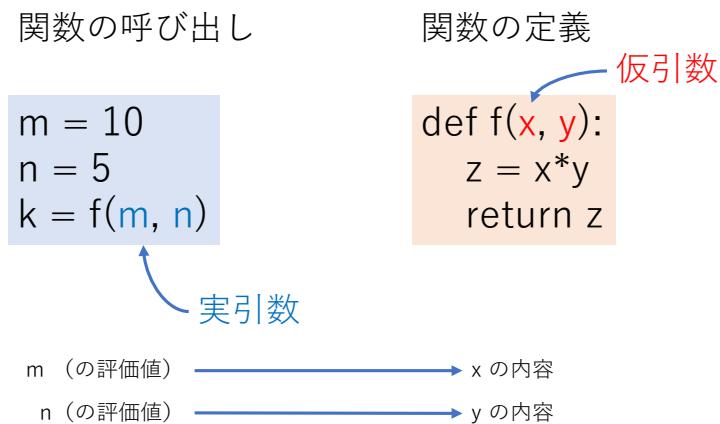


図 7-1 実引数と仮引数

## 7.5 戻り値

関数の計算結果は return 文で返します。

<sup>1</sup> リストの場合は書き換えが可能です。.

- プログラム 7-2 のように関数定義内の複数箇所で `return` 文を書いてしません。ただ、返る場所が複数に分散していると、関数の改修などが難しくなります。
- Python では複数の値を「,」で区切って `return` 文で返すことが可能ですが。呼び出す側は複数の変数を「,」で区切って受け取ります。1つの変数で受け取ると、この変数は複数の返り値の値で構成された「タプル」というデータ形式になります。

## 7.6 前章の例題から

前章では平方根を求めるプログラムと、求めたい数を端末から入力する方法を学び、これらを組み合わせ、繰り返し端末から入力された値について平方根を求めるプログラムの構成に取り組みました。繰り返し端末から入力するプログラムの構成方法としては以下のような2通りが考えられます。

1. 端末から入力を得て、正しい入力が得られた場合にだけ平方根を計算し、入力を繰り返す。
2. 端末から正しい入力を得ることと、その入力についての平方根を求めることを直列に実行することを繰り返す。

これらをプログラミングするなら、上記の記述のように1. の場合は「平方根を求める」、2. なら「端末から正しい入力を得る」「平方根を求める」ということを素直に表現したいところです。それらをそれぞれ `get_positive_numeral()`, `square_root()` という関数で書けるものと想定するとプログラムはそれぞれ以下のように書けるはずです。

### 1. の場合

```
while True:
    端末から入力 x を得て正の数値かどうかを検査する
    if 正の数値:
        r = square_root(x)
        print(r)
```

### 2. の場合

```
while True:
    x = get_positive_numeral()
    r = square_root(x)
    print(r)
```

このように関数として一定の処理を取りまとめることでその処理をカプセル化することができます。関数によるカプセル化のメリットは

- 呼び出す側のプログラムが短く分かりやすくなる。
- 同じ処理をプログラムの別の場所でも使える。
- 関数を利用する側のプログラムと定義する側のプログラムを分離することで、定義の修正などが行いやすくなる。

といったことです。

## 7.7 関数 `square_root()` を実装する

それでは上のうち、平方根の計算プログラム `p5-6.py` を関数にすることで `square_root()` を実装してみましょう。

プログラム 7-3 関数 `square_root()` の実装, p7-3

行	ソースコード	説明
1	#	引数 x を取る関数
2	def square_root(x):	説明文字列(docstring)
3	'''引数xの平方根を求める'''	以下、17 行目までが関数を定義するブロック、 <code>p5-6.py</code> と同じだが、インデントに注意。
4	rnew=x	
5	#	
6	diff=rnew-x/rnew	
7	if diff<0:	
8	diff=-diff	
9	while (diff>1.0E-6):	
10	r1=rnew	
11	r2=x/r1	
12	rnew=(r1+r2)/2	
13	print(r1, rnew, r2)	
14	diff=r1-r2	
15	if diff<0:	
16	diff=-diff	
17	return rnew	計算した値を返り値で戻す
18	#ここからメインプログラム	
19	v=2	
20	r=square_root(v)	
21	print("結果は", r)	

実行結果は以下のようになります。

Python では関数の定義に文字列(docstring)を入れておくとこの関数名を引数に `help` 関数を呼び出すことで説明が表示されます。

```
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
```

```

1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002
結果は 1.414213562373095
>>> help(square_root)
Help on function square_root in module __main__:

square_root(x)
    引数 x の平方根を求める

>>>

```

### 演習 7-3 平方根を求めるプログラムの関数化

繰り返し平方根を求めるプログラムを関数 `square_root()` を定義して利用する形に書き換えなさい。

### 演習 7-4 平方根を求めるプログラムの関数化, その 2

`get_positive_numeral()` も構成し, 繰り返し平方根を求めるプログラムをこれと `square_root()` を利用する形に書き換えなさい。

## 7.8 関数内の変数の扱い

Python では関数内の変数は以下のように扱います。コラム「変数のスコープ」も参考してください。

- ローカル変数：関数内で定義された（代入された）変数は関数内でのみ利用可能で、関数の（毎回の）実行ごとに関数の実行が終了すれば失われます。
- 仮引数もローカル変数として扱われます。
- グローバル変数：関数外で定義されている変数は値を読み取ることのみ可能です。
- グルーバル変数への代入：関数内で `global` 宣言された変数のみ、グローバル変数に代入可能です。

Python で関数内の変数をこのように扱うことにより

- 関数内で一時的に必要な変数は他への影響を考えずに自由に使えます。
- グローバル変数の操作はプログラムが長くなるとプログラムを分かりにくくする要因になるのですが、Python では比較的安全な読み取りは無条件で許可する一方で、書き込みは `global` 宣言でプログラムに明示することを求め安全性と利便性のバランスをとっています。

a = 10 b = 0	a, b はグローバル変数
def f(): global b c = a*a b= c	関数定義 b をグローバル宣言 グローバル変数 a は参照のみ可能 c はローカル変数 グローバル宣言した変数は代入可能
f() print(b, a)	メインプログラム

図 7-2 グローバル変数とローカル変数

## 7.9 関数の利用パターン

数学の関数とは異なり Python の関数の利用パターンはいくつかあります。関数に () 内に記述して与える情報を「引数」、返される値を「返り値」と呼びます。

- 引数を与え、返り値を使う。数学の関数と同様な使い方です。

`y = math.fabs(-2.0)`

- 引数も与えず、返り値も使わない。定型的な命令を実行する場合に使います。

例えば次章の `turtle` の `up()`

- 引数は与えるが返り値は使わない。可変な値を含む命令の実行に使います。

例えば次章の `turtle` の `forward(100)`

- 引数は与えないが返り値を使う。対象の状態を知るのに使います。

例えば次章の `turtle` の `p = pos()`

このほか、「グローバル変数を読み書き」する関数や「リストなど書き換え可能な引数を通じて情報をやりとり」する使い方もありますが、このような使い方は関数の「副作用」と呼ばれ、ソースコード（特に関数を呼び出す側）からは明示化されにくいため、プログラムを分かりにくくするという弊害もあります。

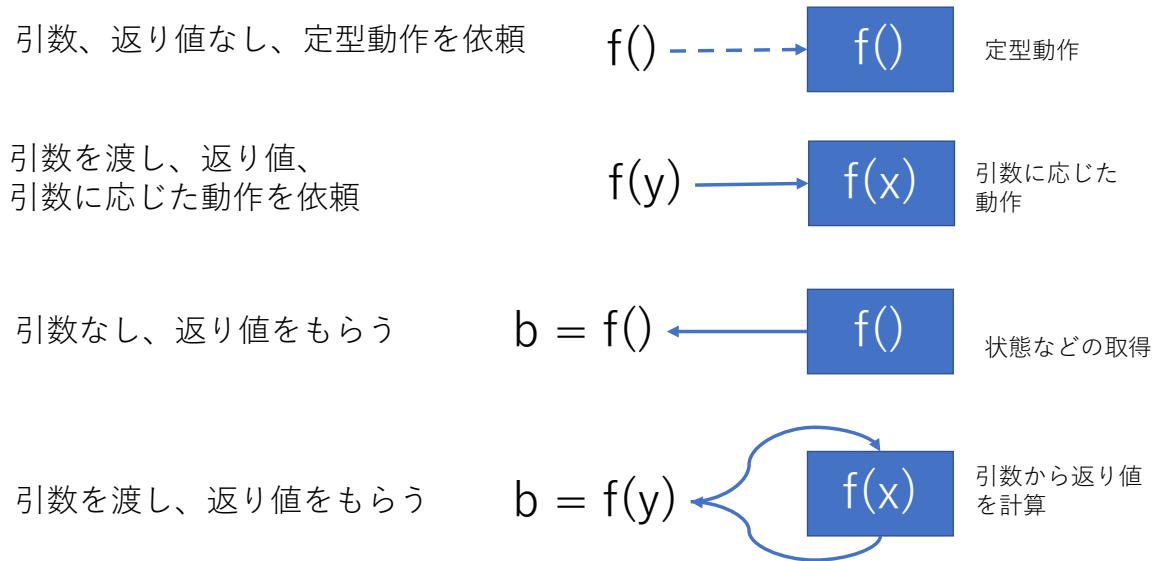


図 7-3 関数の利用パターン

<code>a = 0</code>	グローバル変数
<code>def f():     global a     a = a+1</code>	グローバル変数を操作する関数
<code>def g(x):     x[0] = 0</code>	リスト型の引数の内容を操作する関数
<code>f() print(a) b = [1,2,3] g(b) print(b)</code>	メインプログラム 関数の呼び出しにより グローバル変数の内容が変化 引数のリストの内容が変化

図 7-4 関数呼び出しと「副作用」

## 7.10 関数の呼び出しと関数オブジェクトの引き渡し

`def` 文で定義された関数を実行する際には引数の有無にかかわらず `f()` のようにカッコを付けます。これに対し、後の章で出てくるタートルグラフィックスでのマウスクリック時に実行する関数の指定や、`tkinter` での GUI プログラムでボタンが押された際に実行する関数の指定などでは関数名だけを表記します。このような表記により、関数をその場で実行するのではなく「後で実行する」関数そのものをオブジ

エクトとして引き渡すことができます。以下の例では関数 `f` を関数 `F` の引数として渡して `F` の内で `f` を実行しています。

```
def f():
    print("f says Hello")
# 関数を引数でもらって実行する関数
def F(y):
    print("In F, ", end="")
    y()
# f を実行
f()
f says Hello
# f を F に渡して F を実行
F(f)
In F, f says Hello
```

## 7.11 デフォルト引数とキーワード引数

### 7.11.1 デフォルト引数

Python の関数の定義では引数名のあとに `=` で値を指定することで、引数が与えられない場合に暗黙で使用する値（デフォルト値）を指定することができます。

### 7.11.2 キーワード引数

関数を呼び出す側では、前から順に引数を渡すほかに「引数名=値」という形式（キーワード引数）で、特定の引数だけ値を渡すことができます。

### 7.11.3 例題

```
def f(a, b=2, c=3):
    return a + b + c

f(1,1,1)
3
f(1)
```

6

f(1, c=2)

5

後の章で紹介する `tkinter` のプログラミングでは、数多くの引数があり、キーワード引数で必要なものだけを指定する使い方をします。

## 8. Turtle で遊ぶ

---

### 8.1 本章の学習の目標

1. Turtle を通じて Python でのモジュールの使用法を知る.
2. Turtle を通じて Python のクラス型オブジェクトの使用法を知る.
3. Turtle を使ったグラフィクスの作品作りを通じて、これまでに学んだことを確認するとともに、作品に必要なライブラリの使用などを主体的に学ぶ.

### 8.2 モジュール

Python でのさまざまなライブラリは「モジュール」という形で提供されます。モジュールの利用は、Python スクリプトの中で必要なモジュールをインポート (import) する形で行います。インポートの仕方としては以下のような方法があります。

#### 8.2.1 インポートする場所

通常、プログラムの先頭でインポートします。

#### 8.2.2 モジュール名を指定してインポートする方法

数学関数のモジュール math をインポートする場合を例にすれば

```
import math
```

と書きます。モジュール内の関数や定数などは

```
math.pi
```

のようにモジュール名とドット ‘.’ を pi の前に付ける形で行います。通常はこの形式が用いられます。

#### 8.2.3 別名をつけてインポートする方法

モジュール名を短縮して参照したい場合に使います。例えば、この後に取り上げる tkinter というモジュールはしばしば tk という短縮形を使うように

```
import tkinter as tk
```

という形でインポートし, `tk` という別名で利用します.

### 8.2.4 モジュール内の要素をインポートする方法

本章で使う `turtle` モジュールについては, モジュール内の関数を簡単に利用するために

```
from turtle import *
```

という形でインポートしています. モジュールの関数などをすべて関数名だけで利用可能になりますが, どのモジュールの関数や変数かが分からなくなるので乱用は危険です. コラム「名前空間」も参照してください.

### 8.2.5 モジュールと同じファイル名のプログラムに注意

`import` 命令に出会うと Python は指定されたモジュール名の Python プログラムを予めライブラリなどが収められたフォルダから探索します.

その際, 現在の作業フォルダが最初に探索されるため, 例えば `turtle.py` というファイルを作ってしまうと, これをモジュールのファイルと解釈しエラーになります. モジュール名と同じ名前の Python スクリプトを作業フォルダに作らないことに注意してください.

## 8.3 Turtle —由緒正しき亀さん

タートルグラフィクスは画面上の亀（ロボット）に前進や回転などの命令を与え, その軌跡としてグラフィクスを作成するものです.

これはマサチューセッツ工科大学(MIT)で開発された LOGO という言語に盛り込まれたグラフィクス機能でプログラムの動きを視覚化して学ぶことを意図しています. LOGO の開発者の一人であるシーモア・パパート(Seymour Papert, 1928-2016)は子供たちへのプログラミング教育に取り組んだことで有名ですが, 彼の名言に

今日多くの学校では, 「コンピューターによる学習」というと, コンピューターに子供を教えさせるということを意味する. コンピューターが子供をプログラムするのに使われていると言ってもよい. 私の描く世界では, 子供がコンピューターをプログラムし, そうする過程で最も進んだ強力な科学技術の産物を統御するという実感を得るとともに, 科学, 数学そして知性のモデルを作る学問などからくる深遠な理念と密接な関係を確立するのである.

シーモア・パパート[10], 太字は著者があります。日本では戸塚さんという先生が LOGO を処理するプログラムを自ら作成し小学校での教育に取り組みました[11]。子供でも扱いやすいプログラミング言語として、アラン・ケイ (Alan Kay) らによって開発された Squeak や、MIT で開発された Scratch<sup>1</sup> がありますが、Squeak や Scratch はタートルグラフィクスの機能を持っています。Scratch の導入としてマスコットの猫を走らせるプログラミングの例を用いますがタートルグラフィクスの流れを汲んだものです。Python でも Turtle というモジュールを利用することでタートルグラフィクスが楽しめます。

数値ばかりを扱っていても退屈なので本章ではタートルグラフィクスを通じてこれまで学んだことのおさらいをします。

## 8.4 Python の Turtle モジュール

- Python のタートルグラフィクスは turtle モジュールとして提供されています。
- GUI 環境として tkinter を基盤にしています<sup>2</sup>。
- 1つのタートルを関数呼び出しで操作する手続き指向と複数のタートルを扱えるオブジェクト指向の2種類の使い方が可能です。
- 注意：モジュール名と同じファイル名で **(turtle.py)** で Python プログラムを作らないこと。Python が正しいモジュールを検索できなくなります。

## 8.5 使ってみよう

次の表のプログラムを作成して実行してみてください。

**プログラム 8-1 turtle を使う例**  
**(turtle.py という名前で保存してはいけない, p8-1.py)**

行	ソースコード	説明
1	<code>from turtle import *</code>	<code>turtle</code> で定義されている関数を呼べるようにする。
2	<code>forward(100)</code>	
3	<code>left(90)</code>	
4	<code>forward(100)</code>	
5	<code>left(90)</code>	

<sup>1</sup> Scratch を開発している MIT Media Lab のグループは「Lifelong Kindergarten」（生涯の幼稚園）と名乗っています。素敵なネーミングだと思いませんか？

<sup>2</sup> 本授業で使用している IDLE や次に紹介する GUI 環境と共通です。tkinter は Tcl/Tk という GUI ライブリを用います。

6	forward(100)
7	left(90)
8	forward(100)
9	left(90)
10	done()

終了

forward() はタートルを前進させる, left() は左に回す関数です.

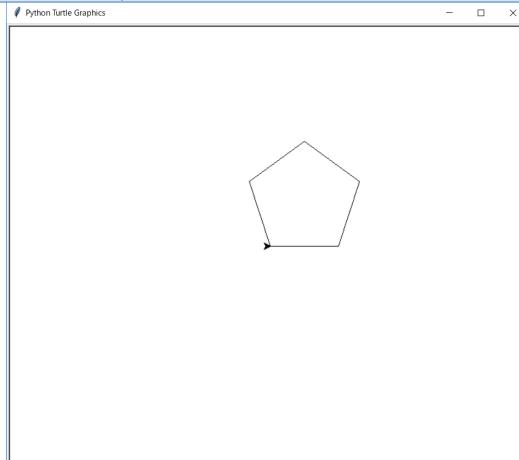
タートルはペンを持っており, ペンが降りている (デフォルト) では, 移動した軌跡が残されます.

### 演習 8-1 正 n 角形を描く

プログラム 8-2 を完成させて正 n 角形を書くプログラムを作成してください.

プログラム 8-2 n 角形を描くプログラム (未完成, p8-2.py)

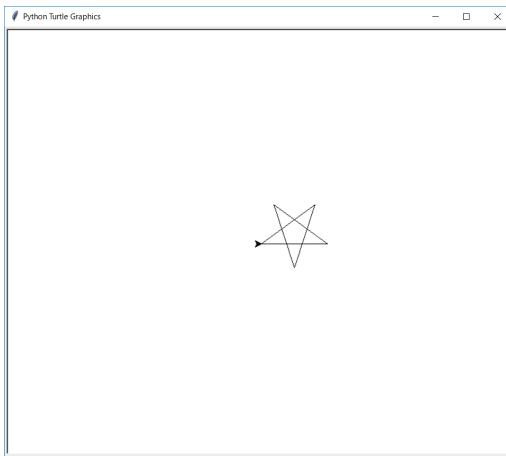
行	ソースコード	説明
1	from turtle import *	turtle で定義されている関数を呼べるようにする.
2	n=5	正 5 角形を描く
3	for i in range(n):	n 回繰り返す
	done()	終了



### 演習 8-2 星形の描画

星形はどうやって描けばいいでしょうか

ヒント： n 角形では turtle の向きが一巡で丁度 1 回転します。星形ではどうでしょう。



### 演習 8-3 正 7 角形, 正 9 角形とそこでの星形

正 7 角形, 正 9 角形とそこでの星形に相当する図形（正多角形の内側を通って頂点を一巡する図形）をタートルで書いてください。

## 8.6 Turtle モジュールの主な関数

以下のような関数が使えます。詳しくは Python ドキュメントの

[24.1. turtle --- タートルグラフィックス](#)

を参照してください。

- `forward(d)`: d だけ前進。`fd(d)` も同じ。
- `back(d)`, `bk(d)`, `backward(d)`: 後退
- `right(a)`, `rt(a)`: 右へ a 度回転
- `left(a)`, `lt(a)`: 左へ a 度回転
- `goto(x, y)`, `setpos(x, y)`, `setposition(x, y)`: 座標 x, y に移動
- `setheading(a)`: 向きを a 度に設定
- `pendown()`, `pd()`, `down()`: 軌跡を描くペンを下ろします。
- `penup()`, `pu()`, `up()`: ペンを上げます
- `position()`, `pos()`: タートルの位置を 2 次元ベクトルとして返します。以下のように 2 つの変数で返り値を受け取ります。

`x, y = pos()`

- `heading()`: タートルの向きを返します
- `isdown()`: ペンが降りていれば `True` を、上がっていれば `False` を返します。

## 8.7 複数のタートルを動かす

### 8.7.1 プログラム例

プログラム 8-3 複数のタートルを動かす (p8-3.py)

行	ソースコード	説明
1	from turtle import *	turtle で定義されている関数を呼べるようにする。
2	t1=Turtle()	1つ目のタートル t1 を作る
3	t2=Turtle()	2つ目のタートル t2 を作る
4	t1.color('red')	t1 の色を赤に
5	t2.color('blue')	t2 の色を青に
6	for i in range(180):	
7	t1.forward(5)	t1 は 5 ステップ前進
8	t2.forward(3)	t2 は 3 ステップ前進
9	t1.left(2)	それぞれ 2 度回転
10	t2.left(2)	
11	done()	終了

### 8.7.2 クラスオブジェクトの利用

この例ではそれぞれのタートルは Turtle クラスのオブジェクトとして生成されます。

1つのタートルは「居る場所（座標）」、「向いている方向」、「ペンが上がっているか、降りているか」、「ペンの色」などの状態をもっています。タートルをプログラミングするためには、これらの状態を知ったり、あるいは状態を変更したりすることが求められます。

クラス型のオブジェクトはこのような操作を記述するのに適した方式です。複数のタートルを使うことで、クラス型オブジェクトの使い方のポイントが理解できると思います。

#### 1) タートルを作る

Turtle クラスのオブジェクトは Turtle() という命令で作成します。クラス型のオブジェクトを生成する関数を特に「コンストラクタ」(constructor タートルを作る (コンストラクト) もの、という意味です<sup>1)</sup> と呼びます。下の例では生成されたオブジェクトを変数 t1 に代入することで、以後、変数 t1 でこのタートルを指定できます。

<sup>1)</sup> コンピュータのプログラミングでは -er などの接尾辞がついた用語によく出会います。これはコンピュータに何かを頼むので擬人的な表現が適しているからです。コンピュータ(computer)という語も機械式の計算機が出現するまでは、計算を担当する人（計算手）を意味していたようです。

```
t1 = Turtle()
```

## 2) タートルを操作する

このタートルの状態を知ったり、状態を変えたりする方法として「メソッド」の呼び出しを行います。メソッドはオブジェクトの変数に「.」とメソッド名（と引数）を書くことで行います。対象がそのオブジェクトであることを除けば関数呼び出しと同じようなものだと考えてください。

```
t1.forward(10)
```

## 3) タートルの状態を知る

タートルの状態を知るには状態を返すメソッドを呼び出して結果を適当な変数に代入するなどすればいいでしょう。

```
x, y = t1.pos()
```

これは以下のように書いてもかまいません

```
(x, y) = t1.pos()
```

## 8.8 作品作りのためのヒント

### 8.8.1 マウスクリックに応答する

プログラム 8-4 タートルグラフィクスでのマウスクリックへの応答

(p8-4.py)

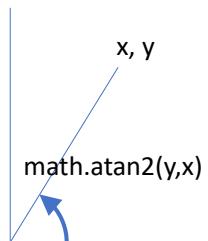
マウスがクリックされたときに実行したい関数定義し、onscreenclick() 関数に定義した関数オブジェクトを渡します。

行	ソースコード	説明
1	from turtle import *	turtle で定義されている関数を呼べるようにする。
2	def come(x,y):	マウスがクリックされたときに実行される関数を定義、引数はクリックしたときのマウスカーソルの位置
3	(xx,yy)=pos()	
4	newxy=((xx+x)/2,(yy+y)/2)	
5	print(x,y)	関数の定義はここまで
6	goto(newxy)	マウスがクリックされたときに呼び出す関数を設定
7	onscreenclick(come)	(come の後に () が無いことに注意)
8	done()	

## 8.8.2 座標を角度に変換する

$x, y$  座標からその方向への角度を求めるために Python では math モジュールに `atan2` という関数を用意しています<sup>1</sup>。引数は縦の座標が前です。返り値はラジアンなので、角度を「度」で設定する `turtle` で使うには、例えば以下のように変換します。

```
import math
y = 2
x = 1
angle = math.atan2(y, x)*180/math.pi
```



## 8.8.3 亂数を使う

ランダムな動きも可視化すると面白いもの一つです。

- Python で乱数を使うには `random` モジュールを使います。
- コラム「乱数」も参照ください
- 乱数を使ったタートルグラフィクスの例を用意しました。`random_turtle.py` 参照ください。
- マウスクリックで停止するようになっています。

---

<sup>1</sup> 角度から座標を計算するには三角関数 `cos` や `sin` を使えばいいのですが、三角関数の逆関数として適当なものがないため、逆正接関数(`atan`)の拡張として `atan2` が導入されています。

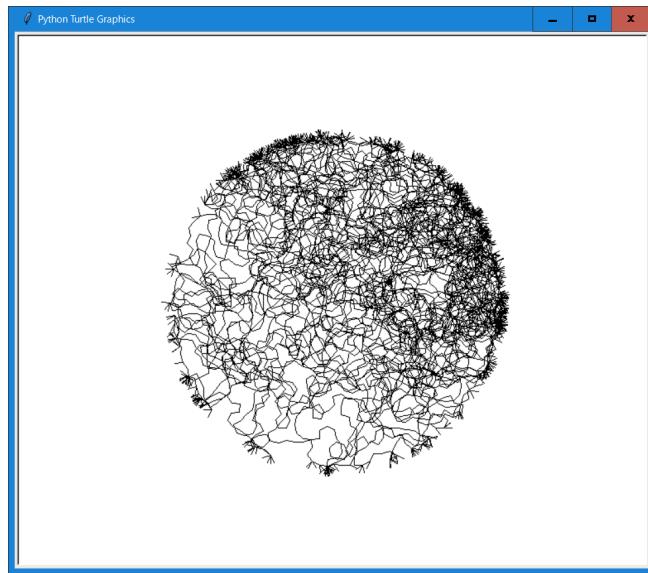


図 8-1 random\_turtle.py の実行結果

#### 8.8.4 フラクタル図形を描く

一部が全体と相似な図形を「フラクタル」と呼びます。フラクタルな図形は描かれたものも、それを描くアルゴリズムも興味深いものです。

- 関数の中で自分自身を呼び出す（再帰）を使う。
- コラム「再帰」も参照してください
- detour.py, turtle-tree.py を参照ください。再起呼び出しの部分を赤字にしています。

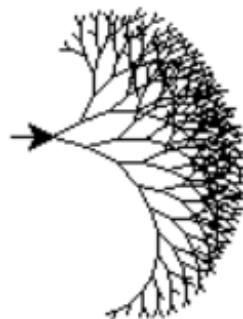


図 8-2 turtle-tree.py の実行結果

## 8.9 Turtle Demo

Python にはタートルグラフィクスのデモプログラムが用意されています。IDLE のメニューから呼び出せます。

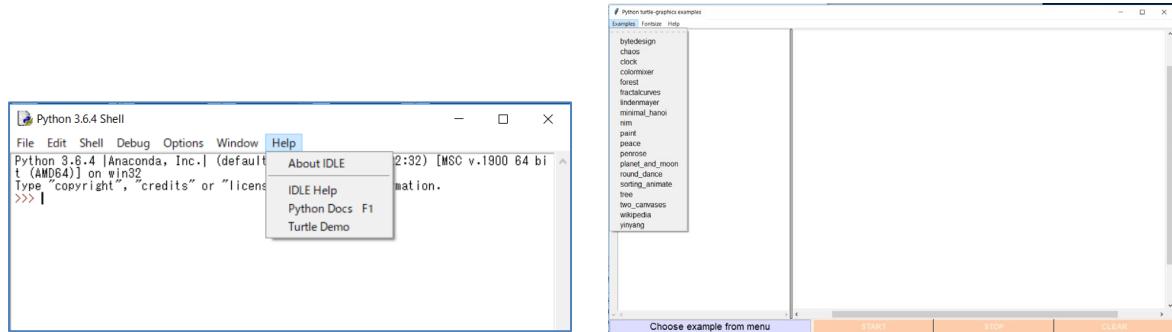


図 8-3 Turtle Demo の実行方法

## 8.10 課題 Turtle の作品制作

ここまで学習した Turtle をを使った作品を制作してください。

- プログラムとスクリーンショット、作成メモを提出すること。
- プログラムは自身でプログラミングすること。
  - ただし、アドバイスを受けることは OK です。作成メモにはその旨、謝辞として記載してください。
  - 提出は添付ファイルとしてください。
- 作成メモには以下を記載すること。
  - 氏名、所属
  - 作品の説明
  - 自身で学習した Python や Turtle グラフィクスの機能
  - 参考にした情報があれば、その書誌情報、Web サイトなら、サイトのタイトル、URL、アクセス日時
  - 支援を得た人がいれば謝辞に支援を得た人と支援内容を書いてください。
  - Word などで作成し、PDF ファイルで提出すること。

## 8.11 スクリーンショットの撮り方

- Windows で特定のウィンドウのスクリーンショットを得るには、以下の手順で進めます。
  - ◆ そのウィンドウを選択しておいて

- ✧ ALT キーをおしながら PRTSC というキーを押してください。
- PRTSC キーだけを押すと画面全体のスクリーンショットが撮られます。
- ✧ この操作でスクリーンショットがクリップボードにコピーされています。
- ✧ これに続けてペイントなど、画像を操作できるプログラムでクリップボードの内容を張り付けて保存すればスクリーンショットが取れます。
- ✧ Mac でのスクリーンショットについて詳しくは[16]参照。

表 4 スクリーンショットの撮り方

デスクトップ全体			特定のウィンドウ	ショットの所在
Windows	Print Screen キー		Alt + Print Screen キー	クリップボード (コピーした状態)
	Win + Print Screen キー		Win + Alt + Print Screen キー	ピクチャのスクリーンショットフォルダ/ビデオのキャプチャフォルダに PNG 形式で保存
Mac	Command + shift + 3		Command + shift + 4 + space	デスクトップに PNG 形式で保存



図 8-4 スクリーンショットを撮るための Windows でのキー操作

## 参考文献

- [14] シーモア・パパート著；奥村貴世子訳：マインドストーム：子供、コンピューター、そして強力なアイデア、未来社（1995）
- [15] 戸塚滝登著：コンピュータ教育の銀河、晚成書房（1995）
- [16] Mac でスクリーンショットを撮る、<https://support.apple.com/ja-jp/HT201361> (2021/5/7 アクセス)

## プログラム 8-5 random\_turtle.py

行	ソースコード
1	from turtle import *
2	import random
3	# 亂数を使うので random モジュールもインポート
4	
5	# 実行を停止するための変数(フラッグ)
6	stop_flag=False
7	
8	# マウスがクリックされたときの関数、引数x,yをとるように
9	# しないといけないが、使わない
10	# 実行停止フラグを True にする
11	
12	def clicked(x,y):
13	global stop_flag
14	stop_flag=True
15	
16	#
17	# マウスがクリックされたときの動作を指定、clicked関数を
18	#呼び出す
19	#
20	onscreenclick(clicked)
21	
22	speed(0)
23	while(not stop_flag):
24	# -90 度から 90 度の範囲でランダムに向きを変える
25	left(random.randint(-90,90))
26	forward(10)
27	# タートルの位置が原点から一定の距離を超えれば、戻る
28	if position()[0]**2+position()[1]**2>200**2:
29	forward(-10)

## プログラム 8-6 detour.py

行	ソースコード
1	from turtle import *
2	def detour(L):
3	if L < 10:
4	forward(L)
5	else:
6	LL = L/3
7	detour(LL)
8	left(60)
9	detour(LL)
10	right(120)
11	detour(LL)
12	left(60)
13	detour(LL)
14	
15	for i in range(6):
16	detour(100)
17	left(60)

## プログラム 8-7 turtle-tree.py

行	ソースコード
1	from turtle import *
2	
3	# 再帰的に木を描く
4	def tree(n):
5	# 引数が 1 以下なら 5 歩すすむ
6	if n<=1:
7	forward(5)
8	else:
9	# 引数は 1 より大きいとき
10	# 引数の値に応じて前進(幹)
11	forward(5*(1.1**n))
12	# 今の位置と向きを記録
13	xx = pos()
14	h = heading()
15	# 左へ 30 度回転
16	left(30)
17	# 大きさ n-2 で木を描く(左の枝)
18	tree(n-2)
19	# ペンを挙げて軌跡を残さない
20	up()
21	# 先に記録した位置(幹の先端)に戻る
22	setpos(xx)
23	setheading(h)
24	# ペンを降ろす
25	down()
26	# 右へ 15 度
27	right(15)
28	# 大きさ n-1 で木を描く(右の枝)
29	tree(n-1)
30	# ペンを上げてもどる
31	up()
32	setpos(xx)
33	setheading(h)
34	# ペンを降ろす
35	down()
36	
37	# 時間がかかるので最も早い描画
38	speed(0)
39	
40	# 大きさ 12 の木を描く
41	tree(12)

## 9. Tkinter で作る GUI アプリケーション(1)

---

### 9.1 本章の学習の目標

この章では `tkinter` を用いた GUI 型のアプリケーションの作成を通じて

- GUI アプリケーションにおけるフレームワークの役割とイベント駆動型プログラミングを理解する。
- GUI アプリケーションにおける MVC アーキテクチャを理解する。
- `tkinter` でのコールバック関数の実装を通じて Python での関数の定義方法をしる。

### 9.2 GUI とイベント駆動型プログラミング

GUI 型のアプリケーションでは、メニュー やボタンなどによるさまざまな操作をユーザ自身が選択して利用します。そして、操作に対してコンピュータが適切に応答することを期待します。

- このようなユーザの操作を「イベント」と呼びます。
- 多くの GUI 型のアプリケーションは GUI 用の「フレームワーク」を利用します。
- フレームワークはマウスやキーボードの操作を監視してイベントを検出し、プログラマーによって設定されたイベント処理用のプログラムを呼び出します。

フレームワークを用いた GUI 型のアプリケーションではプログラマーは主として以下のようないくつかの部分のプログラミングを担います。

- GUI アプリケーションとしてボタンなどを配置する画面の構成
- イベントが発生した際に行う処理の定義

このようなプログラミングをイベントに対する応答を主に記述することからイベント駆動型 (**event-driven**) プログラミングと呼びます。

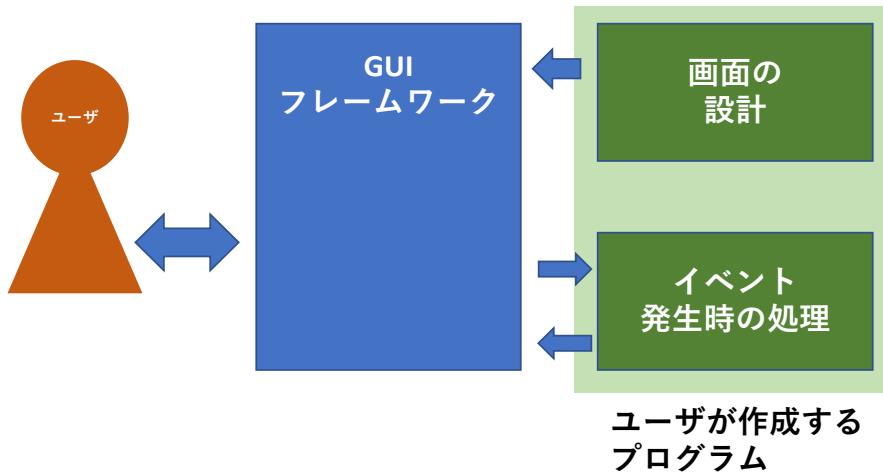


図 9-1 イベント駆動型プログラミングの枠組み

### 9.3 モデルとユーザーインターフェイスの分離

例えば加減乗除を扱う電卓のプログラムを作成することを考えましょう。ユーザがコンピュータに委ねたい仕事は加減乗除が 2 つの数値に対する演算（二項演算）ですから、

- 「第 1 項目の数値」と「第 2 項目の数値」と「適用する演算」を「設定し」、
- 設定された演算を「適用し」、
- 「適用結果」を「得る」

ことです。このうち青色で書いた言葉は数値や演算など操作の対象となる「もの」で「名詞（句）」で示されています。他方で、赤色で書いた言葉は「操作」で、サ变动詞を含む動詞で示されています。これらが「電卓」という仕事の「モデル」を構成します。

一方、このモデルに対して人が関わるために、具体的に人が操作するユーザーインターフェイスが必要になります。パソコン用コンピュータやスマートフォンの電卓アプリケーションのユーザーインターフェイスは具体例の一つですし、Python シェルのように、式をキーボードから入力するという方法も考えられます。また、視覚に障害のある方のためには、音声や点字などでのインターフェイスが求められるでしょう。

「電卓」という応用を中心に考えると、「モデル」は共通で、ユーザーインターフェイスはいろいろ変わるということが言えます。GUI のプログラミングにおいて、このような考え方を表すことばとして「MVC アーキテクチャ」があり、次の M, V, C を分けて考えようというものです。コラム「GUI」も参照してください。

- M: Model, アプリケーションの骨格を与える計算対象のモデル, 基本的に GUI とは独立
- V: View, Model で得られた結果をユーザに示すプログラム, GUI が担う
- C: Control, モデルに対するユーザの操作のためのプログラム, GUI が担う

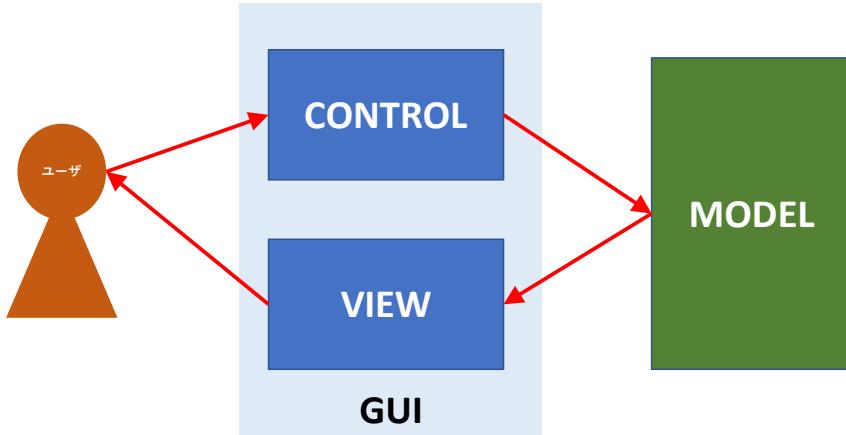


図 9-2 Model-View-Control アーキテクチャ

## 9.4 tkinter

パソコン用コンピュータの基本ソフトウェアとウィンドウ環境としては Windows, macOS, Linux/X-Window など複数のものが使われており, それぞれにウィンドウの描画などは異なった方法で行われます. これらの OS/ウィンドウ環境の差異を吸収し, 共通に使える GUI 用のフレームワークとして Tcl/Tk があります. tkinter はこの Tcl/Tk を Python から利用できるようにしたパッケージです.

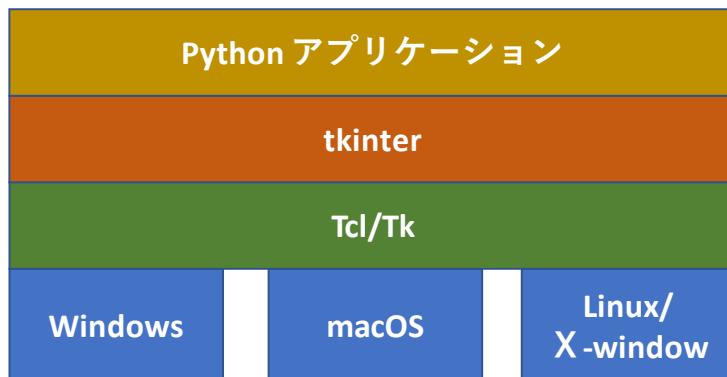


図 9-3 Tkinter のシステム構成

### 9.4.1 tkinter の用語

- widget (ウィジェット) : GUI を構成するボタンなどの部品の総称です。
- コンテナ : widget (群) を格納する入れ物です。
- レイアウト・マネージャ／ジオメトリ・マネージャ : widget の幾何学的配置を調整する仕組みです。
- コールバック (Call back) 関数 : widget が操作されたときに必要な処理を行うために呼び出す関数のことを指します。

## 9.5 tkinter の例題(tkdemo-2term.py)

以下のような加算のみの電卓を考えます。

- 0 ~ 9 の 10 キーと, C(クリア), +(足し算) =(計算) の 13 個のボタンと
- 数値を示す 1 行の文字入出力で構成されます。
- 0~9 は電卓と同様に入力中の数字の最小桁を挿入します。 (入力中の数字を 10 倍して押されたキーの数字を加える)
- C は数値を 0 にします。
- + キーは入力された数値を 2 項演算の第 1 項に登録し, 入力中の数字を 0 にします。
- = キーは入力中の数字を 2 項演算の第 2 項に登録し, 足し算を実行し, その結果を表示するとともに入力中の数字を 0 にします。



プログラム 9-1 はこれの実装例です。

プログラム 9-1 加算のみの電卓(tkdemo-2term.py)

行	ソースコード	説明
1	import tkinter as tk	tkinter を短縮形 tk で参照する形でインポート
2		
3	#_計算機能のための変数とイベント用の関数定義	
4		
5	#_2 項演算のモデル	二項演算を処理するための変数, 関数, GUI に依存しないことに注意
6	#_入力中の数字	
7	current_number_=0	
8	#_第一項	
9	first_term_=0	
10	#_第二項	
11	second_term_=0	
12	#_結果	
13	result_=0	

<pre> 14 15     def do_plus(): 16         """+キーが押されたときの計算動作, 第一項の設定と 17         入力中の数字のクリア""" 18         global current_number 19         global first_term 20 21         first_term = current_number 22         current_number = 0 23 24     def do_eq(): 25         """=キーが押されたときの計算動作, 第二項の設定, 26         加算の実施, 入力中の数字のクリア""" 27         global second_term 28         global result 29         global current_number 30         second_term = current_number 31         result = first_term + second_term 32         current_number = 0 33 34     # 数字キーの Call Back 関数 35     def key1(): 36         key(1) 37 38     def key2(): 39         key(2) 40 41     def key3(): 42         key(3) 43 44     def key4(): 45         key(4) 46 47     def key5(): 48         key(5) 49 50     def key6(): 51         key(6) 52 53     def key7(): 54         key(7) 55 56     def key8(): 57         key(8) 58 59     def key9(): 60         key(9) 61 62     def key0(): 63         key(0) 64 65     # 数字キーを一括処理する関数 66     def key(n): </pre>		ここからはウィジェットの Call Back 関数の定義
		数字キーの処理は数字が異 なるだけで同じ

<pre> 65     global_current_number 66     current_number = current_number * 10 + n 67     show_number(current_number) 68 69     def clear(): 70         global current_number 71         current_number = 0 72         show_number(current_number) 73 74     def plus(): 75         do_plus() 76         show_number(current_number) 77 78     def eq(): 79         do_eq() 80         show_number(result) 81 82     def show_number(num): 83         e.delete(0,tk.END) 84         e.insert(0,str(num)) 85 86     #tkinterでの画面の構成 87 88     root=tk.Tk() 89     f=tk.Frame(root) 90     f.grid() 91 92     #ウィジェットの作成 93 94     b1=tk.Button(f,text='1',command=key1) 95     b2=tk.Button(f,text='2',command=key2) 96     b3=tk.Button(f,text='3',command=key3) 97     b4=tk.Button(f,text='4',command=key4) 98     b5=tk.Button(f,text='5',command=key5) 99     b6=tk.Button(f,text='6',command=key6) 100    b7=tk.Button(f,text='7',command=key7) 101    b8=tk.Button(f,text='8',command=key8) 102    b9=tk.Button(f,text='9',command=key9) 103    b0=tk.Button(f,text='0',command=key0) 104    bc=tk.Button(f,text='C',command=clear) 105    bp=tk.Button(f,text='+',command=plus) 106    be=tk.Button(f,text="=",command=eq) 107 108    #Grid型ジオメトリマネージャによるウィジェットの割付 109 110    b1.grid(row=3,column=0) 111    b2.grid(row=3,column=1) 112    b3.grid(row=3,column=2) 113    b4.grid(row=2,column=0) 114    b5.grid(row=2,column=1) 115    b6.grid(row=2,column=2) 116    b7.grid(row=1,column=0) </pre>	<p>クリアキーの処理 + キーの処理 = キーの処理 数値をエントリーに表示する関数 Tk() でウインドウ作成 Frame コンテナ作成 Frame を割り付け ボタンを表示テキストと Call back 関数を指定して生成 ボタンを grid で位置を指定して Frame に割り付け</p>
---	---

<pre> 117 b8.grid(row=1, column=1) 118 b9.grid(row=1, column=2) 119 b0.grid(row=4, column=0) 120 bc.grid(row=1, column=3) 121 be.grid(row=4, column=3) 122 bp.grid(row=2, column=3) 123 124 # 数値を表示するウィジェット 125 e=tk.Entry(f) 126 e.grid(row=0, column=0, columnspan=4) 127 clear() 128 129 # ここから GUI がスタート 130 root.mainloop() </pre>		<p>文字入力用の Entry ウィジェットを数値表示用に生成、横長に割り付け</p> <p>mainLoop メソッドで GUI の処理に移る</p>
---	--	--

## 9.6 tkinter を用いたプログラムの基本構成

### 1. モジュールのインポート

```
import tkinter as tk          # 短い名称 tk で使えるようにする
```

### 2. Call back 関数の定義

```
def key1():
```

key1 の内容

### 3. ウィンドウの作成

```
root = tk.Tk()
```

### 4. フレームの作成と割り付け。フレームはそのなかにウィジェットを格納するコンテナの一種です。

```
f = tk.Frame(root)          # root を親に Frame を作成し
```

```
f.grid()                   # grid() で割り付け
```

### 5. ウィジェットの作成（ボタン）

```
b1 = tk.Button(f, text='1', command=key1)
```

# f を親に、ボタンを作成、表示文字列は '1'、実行するコマンドは key1

### 6. ウィジェットの作成（エントリー、文字を表示する）

```
e = tk.Entry(f)
```

### 7. レイアウトの指定

```
b1.grid(row=3, column=0)
```

### 8. GUI の実行

```
root.mainloop()
```

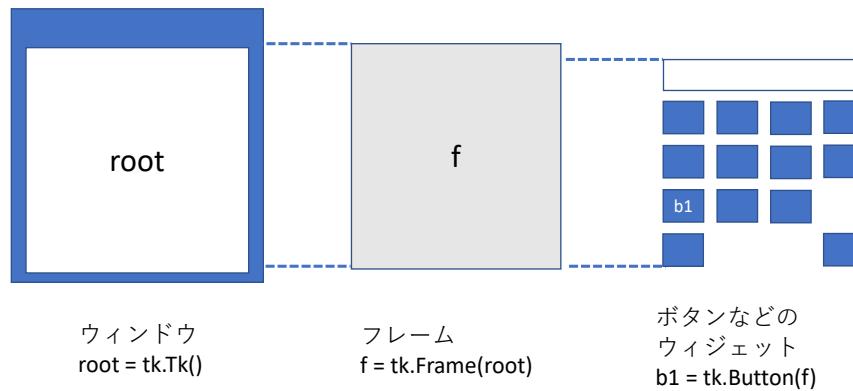


図 9-4 Tkinter でのオブジェクトの関係

## 9.7 grid によるレイアウト

tkinter のウィジェットはレイアウトを管理するレイアウト・マネージャを指定して初めてウィンドウやコンテナに割り付けられます。レイアウト・マネージャには幾通りかありますが、簡単なものとして格子状の位置を与える `grid` があります、以下のように使います。

- 格子状のレイアウトを位置を指定して行う  
`b1.grid(row=3, column=0)`
- いくつかのコラムをまたがる指定も可能  
`e.grid(row=0, column=0, columnspan=4)`

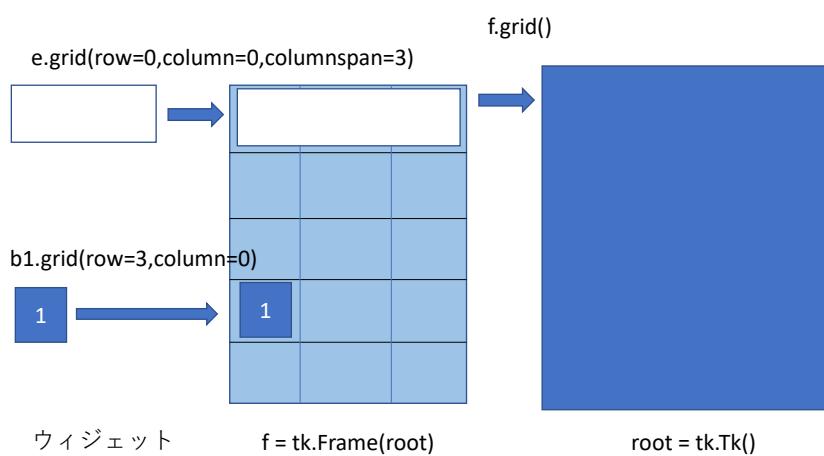


図 9-5 grid によるレイアウト

## 9.8 lambda ( $\lambda$ ) 表現を使った Call Back 関数の記述

先の例では関数 key0() ~ key9() の中身は引数を変えた関数 key() の呼び出しだけです。これは widget の定義において、

```
b1 = tk.Button(f, text='1', command=key1)
```

の中で引数 command=key1 の右辺は「関数オブジェクト」でなければならず

```
b1 = tk.Button(f, text='1', command=key(1)) #これはまちがい
```

と書いてしまうと関数 key() を引数 1 で呼び出した結果の「返り値」が代入されてしまい、関数 key() を引数 1 も指定して call-back 関数として呼び出してもうことになります。他方、key1() という関数はボタン b1 の call-back 専用の関数であり、他には使われることはありません。その意味で、関数 key1() の定義とボタン b1 の call-back 関数の指定を直接結びつけられると便利で key1 という名前は不要になります。

これを実現する記法として Python では lambda 表現<sup>1</sup>というものがあり、その場で関数名をつけずに関数を定義して、変数に代入する手法で、b1 の例では

```
b1 = tk.Button(f, text='1', command=lambda: key1())
```

と書けます。lambda: に続く key(1) がその場限りの関数定義の内容で、実質的に関数 key1 () と同じ内容になります。

### プログラム 9-2 lambda 式を使った引数付きコールバック関数の設定

(tkdemo-2term\_lambda.py)

行	ソースコード	説明
1	import tkinter as tk	
2		
3	#_計算機能のための変数とイベント用の関数定義	
4		
5	#_2項演算のモデル	
6	#_入力中の数字	
7	current_number_=0	
8	#_第一項	
9	first_term_=0	
10	#_第二項	
11	second_term_=0	
12	#_結果	
13	result_=0	
14		
15	def do_plus():	
16	"""+_キーが押されたときの計算動作, 第一項の設定と入力中の数字のクリア"	tkinter を短縮形 tk で参照する形でインポート

<sup>1</sup> 計算機科学で  $\lambda$  計算と呼ばれている理論モデルが名称の由来です。

```
17     global_current_number
18     global_first_term
19
20     first_term_=current_number
21     current_number_=0
22
23 def do_eq():
24     """=キーが押されたときの計算動作、第二項の設定、加算
25     の実施、入力中の数字のクリア"""
26     global_second_term
27     global_result
28     global_current_number
29     second_term_=current_number
30     result_=first_term_+second_term
31     current_number_=0
32
33 # 数字キーを一括処理する関数
34 def key(n):
35     global_current_number
36     current_number_=current_number_*10+n
37     show_number(current_number)
38
39 def clear():
40     global_current_number
41     current_number_=0
42     show_number(current_number)
43
44 def plus():
45     do_plus()
46     show_number(current_number)
47
48 def eq():
49     do_eq()
50     show_number(result)
51
52 def show_number(num):
53     e.delete(0,tk.END)
54     e.insert(0,str(num))
55
56 # tkinterでの画面の構成
57
58 root_=tk.Tk()
59 f_=tk.Frame(root)
60 f.grid()
61
62 # ウィジェットの作成
63 b1_=tk.Button(f,text='1',command=lambda:key(1))
64 b2_=tk.Button(f,text='2',command=lambda:key(2))
65 b3_=tk.Button(f,text='3',command=lambda:key(3))
66 b4_=tk.Button(f,text='4',command=lambda:key(4))
67 b5_=tk.Button(f,text='5',command=lambda:key(5))
68 b6_=tk.Button(f,text='6',command=lambda:key(6))
```

```

69 b7=tk.Button(f,text='7',command=lambda:key(7))
70 b8=tk.Button(f,text='8',command=lambda:key(8))
71 b9=tk.Button(f,text='9',command=lambda:key(9))
72 b0=tk.Button(f,text='0',command=lambda:key(0))
73 bc=tk.Button(f,text='C',command=clear)
74 bp=tk.Button(f,text='+',command=plus)
75 be=tk.Button(f,text="=",command=eq)
76
77 #Grid型ジオメトリマネージャによるウィジェットの割付
78
79 b1.grid(row=3,column=0)
80 b2.grid(row=3,column=1)
81 b3.grid(row=3,column=2)
82 b4.grid(row=2,column=0)
83 b5.grid(row=2,column=1)
84 b6.grid(row=2,column=2)
85 b7.grid(row=1,column=0)
86 b8.grid(row=1,column=1)
87 b9.grid(row=1,column=2)
88 b0.grid(row=4,column=0)
89 bc.grid(row=1,column=3)
90 be.grid(row=4,column=3)
91 bp.grid(row=2,column=3)
92
93 #数値を表示するウィジェット
94 e=tk.Entry(f)
95 e.grid(row=0,column=0,columnspan=4)
96 clear()
97
98 #ここからGUIがスタート
99 root.mainloop()

```

## 9.9 ウィジェットの体裁の調整

ボタンなどのウィジェットの色や大きさ、フォントなどの体裁を調整する方法はいくつかあります。

### 9.9.1 作成時に引数で設定する方法

ウィジェット生成する際の引数で指定します。キーワード引数を使うと便利です。

- `font=('Helvetica', 14)` 文字フォントと大きさ
- `width=2` ウィジェットの大きさ  
(ボタンでは文字数)
- `bg = '#fffffc0'` 背景色の指定、RGB でそれぞれ 16 進数 2 桁

00 暗い, ff 明るい

なお, Mac では現状 bg の設定で色が変わりません. bg の代わりに highlightbackground という引数を設定するとボタン自体の色は変わりませんがボタン周辺の色が変わるため, 演習時はこれで代用して下さい.

表 9-1 tkinter での色指定

表記	Red	Green	Blue	色
'#ffffff'	ff	ff	ff	白
'#000000'	00	00	00	黒
'#ff0000'	ff	00	00	赤
'#00ff00'	00	ff	00	緑
'#0000ff'	00	00	ff	青

## 9.9.2 生成されたウィジェットの体裁を調整する（その1）

生成されたウィジェットの体裁を実行中に変更するにはそのウィジェットの configure() メソッドを用います. 引数の与え方は生成時のキーワード引数と同じです. 例えば b を Button ウィジェットとすると

```
b.configure(size=2)
のように設定します.
```

現在の値を参照するには cget() メソッドの引数に属性を与えて呼び出します. 属性を文字列で与えることに注意してください.

```
b.cget("size")
```

## 9.9.3 生成されたウィジェットの体裁を調整する（その2）

ウィジェットの体裁は以下のような方法でも設定や参照ができます. [] 内の指定は文字列（””などで囲む）でなければならないことに注意してください. 設定は代入で行います.

```
b["size"] =2
print(b["size"])
```

### 演習 9-1 Tkinter での加算電卓の作成

プログラム 9-1 もしくはプログラム 9-2 について実際にプログラムを作成し, 動

作を確認してください。

### 演習 9-2 ウィジェットの体裁の調整

- 足し算電卓のフォントサイズ、 ウィジェットの色を以下のように設定してください。 背景色を Frame は '#ffffcc0'（薄黄色）、 数字キーは白、 クリアキーは赤、 +, = キーは緑にする。
- ボタンの大きさは 2（文字分）にする。
- ボタンとエントリーのフォントとサイズは('Helvetica', 14)にする。

### 演習 9-3 電卓の四則演算への拡張（力試し）

足し算電卓を 4 則演算が可能なように拡張してください。 ただし以下に留意すること。

- ボタンの配置は適宜検討すること。
- 割り算は 0 で割るエラーが発生する可能性があるので、 第 2 項の数値が 0 の場合は何もしないか、 エラーを表示する。
- 割り算の小数点以下は切り捨てる。 Python で整数商を求める演算子は「//」です。

ヒント： プログラムの拡張作業は以下の 2 つになります。

- 四則演算を指定するためのボタンウィジェットの追加  
(こちらはそれほど難しくないでしょう)
- 演算ボタンや「=」ボタンが押されたときのコールバック関数の設定  
以下をヒントに考えてください。
  - 電卓では + などの演算キーが押されたときにはその演算は行われず、 = キーが押されたときに実行されます。 このため、 加減乗除の演算キーが押されたときには、 どの演算を行うべきかを実行の際まで変数に記憶しておきます。 例えば operation という変数を設けて、 加算なら 1、 減算なら 2、 乗算なら 3、 除算なら 4 などの値に設定します。
  - = キーが押されたときに、 記憶していた演算に応じて動作を変える必要があります。 例えば以下のようになるでしょう。

```

if operation == 1:
    加算を実行するブロック
elif operation == 2:
    減算を実行するブロック
elif operation == 3:
    乗算を実行するブロック

```

```

    乗算を実行するブロック
else:
    除算を実行するブロック

```

#### 演習 9-4 ウィジェットのリストでの管理（力試し）

プログラム 9-1 やプログラム 9-2 では Button ウィジェットを数多く使います。これをリストの要素として扱うことを考えてください。またリストとして扱うことで for 文を使った操作ができないかも考えてください。

ただし、プログラム 9-2 の lambda 式（例えば `command=lambda:key(1)`）では関数 `key(1)` の引数 1 は数値として与えていますが、これを lambda 式の定義するさいに変数（例えば `i`）の値を評価して書くことはできません。引数のある lambda 式の記法を使い、引数の暗黙知として以下のように与えます。

```
command=lambda x = i:key(x)
```

#### 演習 9-5 実際の電卓との差異

作成したプログラムと実際の電卓（や電卓アプリ）との動作の違いを探ってください。例えば = キーの代わりに + などの演算キーを押した場合の動作など。実際の製品がしっかり設計されていることが分かると思います。

## 9.10 tkinter の終わり方

`tkinter` を用いたアプリケーションでは `mainloop()` を呼び出すと、ユーザの操作を待って、call back 関数を呼び出す無限ループになります。ウィンドウの終了ボタンで終了する以外の終了方法は以下のようになります。

- `mainloop()` から脱出するには、何かの call back 関数の中で `tk.Tk()` で作成したオブジェクト（例えば `root`）の `quit()` メソッドか `destroy()` メソッドを呼び出します。これらの動作の違いは以下のようになります。
  - `quit()`: ループは脱出しますが、ウィンドウやウィジェットは残ります。
  - `destroy()`: ループを脱出し、ウィンドウやウィジェットそのものをなくします。

## 9.11 Frame クラスを拡張する方式での実装法

先のプログラムでは Frame と Button などのウィジェットは別に構成していましたが、tkinter の実装例では Frame を拡張したクラスとして、その初期化の中でウィジェットを生成するプログラムをしばしば見かけます。ここでは、この方式での実装例を示します。クラスそのものについては後の章の解説を参照してください。

プログラム 9-3 Frame クラスを拡張する tkinter の実装法

(`tkdemo-2term_frame_extention.py`)

行	ソースコード	説明
1	<code>import tkinter as tk</code>	
2		
3	<code>#_計算機能のための変数とイベント用の関数定義</code>	
4	<code>#_Frame_のサブクラスを使った実装例</code>	
5		
6	<code>#_2 項演算のモデル</code>	
7	<code>#_入力中の数字</code>	
8	<code>current_number_=0</code>	
9	<code>#_第一項</code>	
10	<code>first_term_=0</code>	
11	<code>#_第二項</code>	
12	<code>second_term_=0</code>	
13	<code>#_結果</code>	
14	<code>result_=0</code>	
15		
16	<code>def do_plus():</code>	
17	<code>    """+_キーが押されたときの計算動作,_第一項の設定と入力中の数字のクリア"</code>	
18	<code>    global current_number</code>	
19	<code>    global first_term</code>	
20	<code>    first_term_=current_number</code>	
21	<code>    current_number_=0</code>	
22		
23	<code>def do_eq():</code>	
24	<code>    "=_キーが押されたときの計算動作,_第二項の設定,_加算の実施,_入力中の数字のクリア"</code>	
25	<code>    global second_term</code>	
26	<code>    global result</code>	
27	<code>    global current_number</code>	
28	<code>    second_term_=current_number</code>	
29	<code>    result_=first_term_+second_term</code>	
30	<code>    current_number_=0</code>	
31	<code>#</code>	
32	<code>#_tk.Frame_を継承した_MyFrame_というクラスを作り</code>	
33	<code>#_その中でウィジェットやコールバック関数_(メソッド)_を</code>	
34	<code>#_設定する._tkinter_をつかう定番</code>	
35	<code>#</code>	
36	<code>class MyFrame(tk.Frame):</code>	

```

37  #
38  #__init__ はクラスオブジェクトを作る際の初期化メソッド
39  # アンダースコアは前後それぞれ 2つづつ
40  def __init__(self, master=None):
41      super().__init__(master)
42  #あとで参照しないウィジェットの作成、ローカル変数
43      b1=tk.Button(self, text='1', command=lambda:self.key(1))
44      b2=tk.Button(self, text='2', command=lambda:self.key(2))
45      b3=tk.Button(self, text='3', command=lambda:self.key(3))
46      b4=tk.Button(self, text='4', command=lambda:self.key(4))
47      b5=tk.Button(self, text='5', command=lambda:self.key(5))
48      b6=tk.Button(self, text='6', command=lambda:self.key(6))
49      b7=tk.Button(self, text='7', command=lambda:self.key(7))
50      b8=tk.Button(self, text='8', command=lambda:self.key(8))
51      b9=tk.Button(self, text='9', command=lambda:self.key(9))
52      b0=tk.Button(self, text='0', command=lambda:self.key(0))
53      bc=tk.Button(self, text='C', command=self.clear)
54      bp=tk.Button(self, text='+', command=self.plus)
55      be=tk.Button(self, text="=", command=self.eq)
56
57  # Grid 型ジオメトリマネージャによるウィジェット割付
58      b1.grid(row=3, column=0)
59      b2.grid(row=3, column=1)
60      b3.grid(row=3, column=2)
61      b4.grid(row=2, column=0)
62      b5.grid(row=2, column=1)
63      b6.grid(row=2, column=2)
64      b7.grid(row=1, column=0)
65      b8.grid(row=1, column=1)
66      b9.grid(row=1, column=2)
67      b0.grid(row=4, column=0)
68      bc.grid(row=1, column=3)
69      be.grid(row=4, column=3)
70      bp.grid(row=2, column=3)
71
72  # 他のメソッドで参照する数値を表示するウィジェット,
73  # クラスオブジェクトの変数として作成, 頭に self. がつく
74  #
75      self.e=tk.Entry(self)
76      self.e.grid(row=0, column=0, columnspan=4)
77  # クラスの定義では
78  # メソッドの最初の引数は self, 中でクラスオブジェクトの変数,
79  # メソッドは self をつけて参照
80  #
81  def key(self, n):
82      global current_number
83      current_number=current_number*n+10
84      self.show_number(current_number)
85
86  def clear(self):
87      global current_number
88      current_number=0
89      self.show_number(current_number)

```

```
90
91     def plus(self):
92         do_plus()
93         self.show_number(current_number)
94
95     def eq(self):
96         do_eq()
97         self.show_number(result)
98
99     def show_number(self, num):
100        self.e.delete(0, tk.END)
101        self.e.insert(0, str(num))
102
103    #
104    # ここからメインプログラム
105    #
106    root = tk.Tk()
107    f = MyFrame(root)
108    f.pack()
109    f.mainloop()
```

拡張したクラスを使う

## 参考文献

Tkinter については以下の資料のほか、さまざまな解説記事がインターネット上に公開されています。Python 2 と Python 3 で tkinter の使い方が若干異なっています。例えば import するモジュールが Python 2 では Tkinter であるのに対し、Python 3 では tkinter であるなどです。記事などを参照する際には注意してください。

- [17] Tkinter 8.5 reference: a GUI for Python  
<https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>

# 10. Tkinter で作る GUI アプリケーション(2)

## 10.1 本章の学習の目標

本章ではアナログ時計を tkinter で作成することを通じて

- アニメーションなど自律的に動くプログラムと GUI との共存の方法と
- Canvas ウィジェットを用いたグラフィックスの描画について学びます。

## 10.2 自律的に動作するプログラムと GUI との衝突

tkinter など GUI のフレームワークはユーザの操作を観測してマウスクリックなどイベントが発生すると設定されたコールバック関数に処理を委ねます。その際、コールバック関数の処理はすみやかに終了することを想定しており、終了をまって再びイベントの発生の観測に戻ります。

他方で、アニメーションなどプログラム自身が継続的動く場合、これをコールバック関数で呼んでしまうと、イベントの観測が停止してしまいます。

tkinter ではこれら両方のニーズを調整する方法として、一定時間後に指定されたコールバック関数を実行する `after` というメソッドが用意されています。アニメーションなどの継続的な処理を一定時間ごとに行う処理としてコールバック関数で実行し、設定した時間後の処理を `after` メソッドで tkinter に登録してコールバック関数を終えることで、GUI のイベント観測ループを長い時間止めないようにするのです。

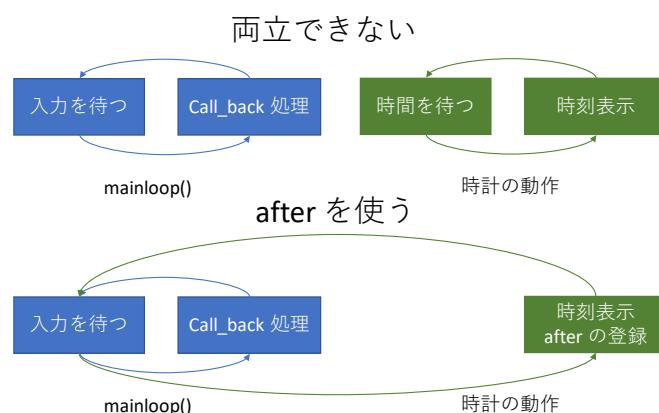


図 10-1 tkinter での `after` の利用

なお、シミュレーションなど、計算時間そのものを要する応用にはこの方法はあまり適しません。プログラムを並行動作させるスレッド(thread)などの考え方、ライブラリの使用を検討する必要があります。

## 10.3 tkinter を用いたアナログ時計プログラム

ここでは

図 10-2 に示すようなアナログ時計を作成します。時針、分針、秒針を表示するほか、日付の表示をボタンでオンオフできるものです。時計の針先位置の計算は図 10-3 を参考にしてください。

以下のプログラムは Frame クラスを拡張する方法で実装されています。長くなる行がありますが、以下のリストで行番号がついていないところは、長い行を折り返している箇所です。入力の際に注意してください。



図 10-2 作成するアナログ時計

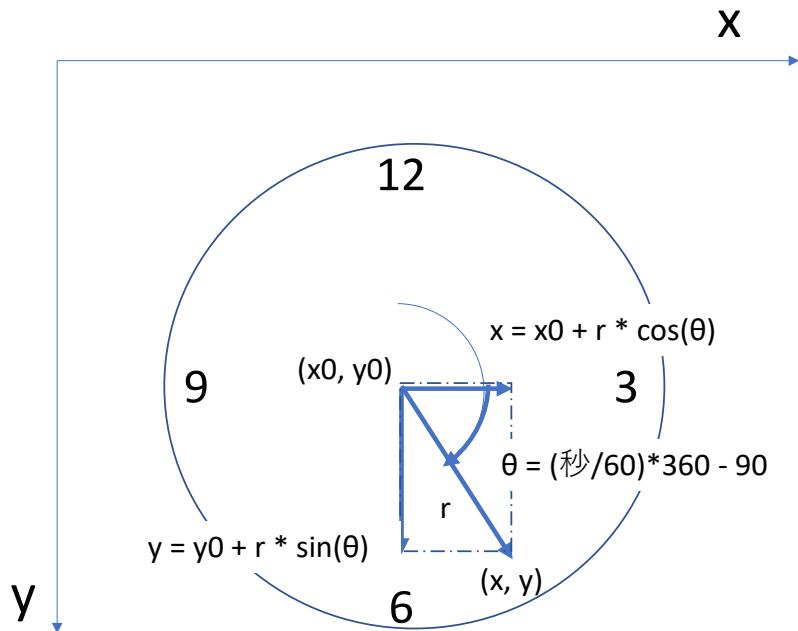


図 10-3 時計の針先位置の計算

### 10.3.1 ソースコード

プログラムは少し複雑ですので、最初にボタンによる日付表示のオンオフのないものを示し、次にこれにボタンを付加したものを示します。

プログラム 10-1 `tkinter` でのアナログ時計（ボタンなし, `tkdemo_simple_clock.py`）

行	ソースコード	説明
1	#	
2	#_tkinter_canvas_を使ったアナログ時計_ ボタンなし	
3	#	
4	import_tkinter_as_tk	時刻を扱うため time をインポート
5	import_math	
6	import_time	
7	#	
8	#_Frame_を拡張したクラス	
9	#	
10	class_MyFrame(tk.Frame):	アンダースコアは 2つづつ
11	def__init__(self, master=None):	
12	super().__init__(master)	
13	#	
14	#_キャンバスの作成	

```

16  #
17      self.size=200
18      self.clock=tk.Canvas(self, width=self.size,
19      height=self.size, background="white")
20      self.clock.grid(row=0, column=0)
21  #
22  # 文字盤の描画
23  #
24      self.font_size=int(self.size/15)
25      for number in range(1,12+1):
26          x=self.size/2+math.cos(math.radians(number*360/
27          12-90))*self.size/2*0.85
28          y=self.size/2+math.sin(math.radians(number*360/
29          12-90))*self.size/2*0.85
30          self.clock.create_text(x,y, text=str(number),
31          fill="black", font=(" ", 14))
32  #
33  # 時刻の経過確認などの動作のためのインスタンス変数
34  #
35  # 変化する画面の描画
36  #
37  def display(self):
38      #
39      # 秒針の描画
40      #
41      self.sec=time.localtime().tm_sec
42      angle=math.radians(self.sec*360/60-90)
43      x0=self.size/2-math.cos(angle)*self.size/2*0.1
44      y0=self.size/2-math.sin(angle)*self.size/2*0.1
45      x=self.size/2+math.cos(angle)*self.size/2*0.75
46      y=self.size/2+math.sin(angle)*self.size/2*0.75
47      #
48      # 前の描画をタグで検索して消してから描画
49      #
50      self.clock.delete("SEC")
51      self.clock.create_line(x0,y0,x,y, width=1, fill="red",
52      tag="SEC")
53      #
54      # 分針、時針の描画、1分毎、時針は分まで考慮
55      #
56      x0=self.size/2
57      y0=self.size/2
58      self.min=time.localtime().tm_min
59      angle=math.radians(self.min*360/60-90)
60      x=self.size/2+math.cos(angle)*self.size/2*0.65
61      y=self.size/2+math.sin(angle)*self.size/2*0.65
62      self.clock.delete("MIN")
63      self.clock.create_line(x0,y0,x,y, width=3, fill="blue",
64      tag="MIN")

```

描画用のウィジエット

```

63         self.hour_=time.localtime().tm_hour
64         x0_=self.size/2
65         y0_=self.size/2
66         angle_=math.radians((self.hour%12+self.min/60)*360/12
67             -90)
68         x_=self.size/2+math.cos(angle)*self.size/2*0.55
69         y_=self.size/2+math.sin(angle)*self.size/2*0.55
70         self.clock.delete("HOUR")
71         self.clock.create_line(x0,y0,x,y, width=3, fill="green"
72             , tag="HOUR")
73         #
74         # 日付の描画, 秒が変わるか, ボタンが押されたとき
75         #
76         x_=self.size/2
77         y_=self.size/2+20
78         text_=time.strftime('%Y/%m/%d %H:%M:%S')
79         self.clock.delete("TIME")
80         self.clock.create_text(x,y, text=text, font=(" ", 12),
81             fill="black", tag="TIME")
82         #
83         # 100ミリ秒後に再度呼び出す
84         #
85         self.after(100, self.display)
86
87         root_=tk.Tk()
88         f_=MyFrame(root)
89         f.pack()
90         f.display()
91         root.mainloop()

```

ここからメイン  
プログラム

最初に display  
を呼び出してお  
く

### プログラム 10-2 tkinter でのアナログ時計（ボタンあり, tkdemo\_clock\_with\_button.py）

行	ソースコード	説明
1	#	

```

16  #
17  self.size_=200
18  self.clock_=tk.Canvas(self, width=self.size,
19  height=self.size, background="white")
20  self.clock_.grid(row=0, column=0)
21  #
22  # 文字盤の描画
23  #
24  self.font_size_=int(self.size/15)
25  for number in range(1,12+1):
26      x_=self.size/2+math.cos(math.radians(number*360/12-
27      90))*self.size/2*0.85
28      y_=self.size/2+math.sin(math.radians(number*360/12-
29      90))*self.size/2*0.85
30      self.clock_.create_text(x,y, text=str(number),
31      fill="black", font=(" ",14))
32  #
33  # 日付表示をオンオフするボタンの作成
34  #
35  self.b_=tk.Button(self, text="Show Date",
36  font=(" ",14), command_=self.toggle)
37  self.b_.grid(row=1, column=0)
38  #
39  # 時刻の経過確認などの動作のためのインスタンス変数
40  #
41  self.sec_=time.localtime().tm_sec
42  self.min_=time.localtime().tm_min
43  self.hour_=time.localtime().tm_hour
44  self.show_date_=False
45  #
46  # ボタンが押されたときの callback
47  #
48  def toggle(self):
49      if self.show_date:
50          self.b_.configure(text="show date")
51      else:
52          self.b_.configure(text="hide date")
53      self.show_date_=not self.show_date
54  #
55  # 変化する画面の描画
56  #
57  def display(self):
58      #
59      # 秒針の描画
60      #
61      self.sec_=time.localtime().tm_sec
62      angle_=math.radians(self.sec*360/60-90)
63      x0_=self.size/2-math.cos(angle)*self.size/2*0.1
64      y0_=self.size/2-math.sin(angle)*self.size/2*0.1
65      x_=self.size/2+math.cos(angle)*self.size/2*0.75
66      y_=self.size/2+math.sin(angle)*self.size/2*0.75

```

描画用のウ  
ィジェット日付表示す  
るかどうか  
ボタンのテ  
キストを切  
り替え  
日付表示す  
るかどうか  
を反転

```

64      #
65      # 前の描画をタグで検索して消してから描画
66      #
67      self.clock.delete("SEC")
68      self.clock.create_line(x0,y0,x,y, width=1, fill="red",
69      tag="SEC")
70      #
71      # 分針、時針の描画、1分毎、時針は分まで考慮
72      #
73      self.min_=time.localtime().tm_min
74      x0_=self.size/2
75      y0_=self.size/2
76      angle_=math.radians(self.min*360/60_-_90)
77      x_=self.size/2+_math.cos(angle)*self.size/2*0.65
78      y_=self.size/2+_math.sin(angle)*self.size/2*0.65
79      self.clock.delete("MIN")
80      self.clock.create_line(x0,y0,x,y, width=3, fill="blue",
81      tag="MIN")
82      self.hour_=time.localtime().tm_hour
83      x0_=self.size/2
84      y0_=self.size/2
85      angle_=math.radians((self.hour%12+self.min/60)*360/12_-_
86      90)
87      x_=self.size/2+_math.cos(angle)*self.size/2*0.55
88      y_=self.size/2+_math.sin(angle)*self.size/2*0.55
89      self.clock.delete("HOUR")
90      self.clock.create_line(x0,y0,x,y, width=3, fill="green",
91      tag="HOUR")
92      #
93      # 日付の描画
94      #
95      x_=self.size/2
96      y_=self.size/2+_20
97      text_=time.strftime(' %Y/%m/%d %H:%M:%S ')
98      self.clock.delete("TIME")
99      if self.show_date:
100         self.clock.create_text(x,y, text=text, font=(" ",12),
101         fill="black", tag="TIME")
102         #
103         # 100ミリ秒後に再度呼び出す
104         #
105         self.after(100, self.display)
106
107         root_=tk.Tk()
108         f_=MyFrame(root)
109         f.pack()
110         f.display()
111         root.mainloop()

```

日付表示するときのみ描画

ここからメインプログラム

### 10.3.2 このプログラムのポイント

- モジュールのインポート `tkinter` のほか、時刻を扱うため `time` を、三角関数を利用するため `math` モジュールをインポートしています。
- `Frame` を拡張した `MyFrame` クラスを定義し、そのなかでウィジェットの作成、割り付け、コールバック関数の定義をしています。
  - `MyFrame` クラス `__init__()` メソッドはクラスのオブジェクトを生成する際に自動的に呼び出されるメソッドです。この中で必要なウィジェットを作成しています。アンダースコアは前後、それぞれ 2 つづつ。
  - ✧ 描画に用いる `Canvas` ウィジェットを生成しています
  - ✧ `Canvas` ウィジェットの `create_text()` メソッドを呼び出す形で文字盤を描画しています。
  - ✧ 時刻の文字での表示の有無を切り替えるボタンを生成しています。（ボタン付きのみ）
  - ✧ 時刻の経過、表示の切り替えなどのためのインスタンス変数を確保し、`time.localtime()` 関数の秒、分、時などで値を設定しています。
  - ボタンが押されたときのコールバック関数の定義。ボタン(b)の表示文字を `b.configure()` メソッドを呼び出して切り替えたり、状態を表す変数を設定したりしています。（ボタン付きのみ）
  - 時計の文字盤を描画するメソッド。
    - ✧ `Canvas` での描画は描画したものに「タグ」をつけておくと、それで後から消去できます。古い描画（時計の針）などをそれではまず消去します。
    - ✧ 時計の針の座標を時刻から三角関数で変換して計算し、`create_line()` メソッドで描画しています。
    - ✧ このメソッドの最後で `after` メソッドで 100 ミリ秒後に、このメソッド自身の呼び出しを設定することで継続的な時計の描画を行います。
- 最後にメインプログラムがあります。`Tk()` メソッドでウィンドウを作り、`MyFrame` クラスのオブジェクトを生成し、初回分の描画を `f.display()` で行ったあと、`mainloop()` でプログラムの制御を `tkinter` に渡します。

#### 演習 10-1 アナログ時計で使用するメソッドなどの確認

このプログラムで呼び出している `time` モジュール、`math` モジュール、`tkinter` の `Canvas` クラスのメソッドなどをリストアップし、メソッドの内容などを Python のオンラインマニュアルなどで確認しなさい。

### 演習 10-2 アナログ時計の改造

アナログ時計のプログラムについて以下の改造を加えなさい。

1. 日付の表示について、日付と時刻ではなく、日付と午前、午後を表示するようにしてください。
2. ボタンをもう一つ追加し、秒針の表示をする、しない、を切り替えるようにしてください。

ヒント：メソッド `self.toggled` と変数 `self.show_date` の役割を確認して、秒針の表示について同じようなことを行うにはどうすればいいかを考えてください。

### 演習 10-3 表示の改善

このプログラムでは 100 ミリ秒ごとに時計の針や日付を再描画していますが、変化のないものまで再描画しています。秒針や日付は秒単位で、分針、時針は分単位でしか位置が変わりません。前回描画した時から変化があったときにのみ描画するようにするにはどうすればいいか考えてください。また、その際、画面が最初に表示されたときやボタンが押されたときには 1 秒程度、表示が遅れてしまうことがないようにするにはどうすればいいか考えてください。

## 10.4 変数を介した動作の協調

プログラム 10-2 では、日付の表示をオンオフするボタンが押されたときには `call back` 関数として設定された `toggle()` メソッドの中で変数 `show_date` の値を切り替えているだけです。他方、タイマ仕掛けで継続的に動作するメソッド `display()` では、この `show_date` 変数を見て、日付の表示のオンオフを切り替えてています。

このように独立して動くメソッド間で動作を協調させるために変数を介して設定を使えることが行われています。`show_date` のような変数は旗の上げ下げに見立ててフラッグと呼ばれます。

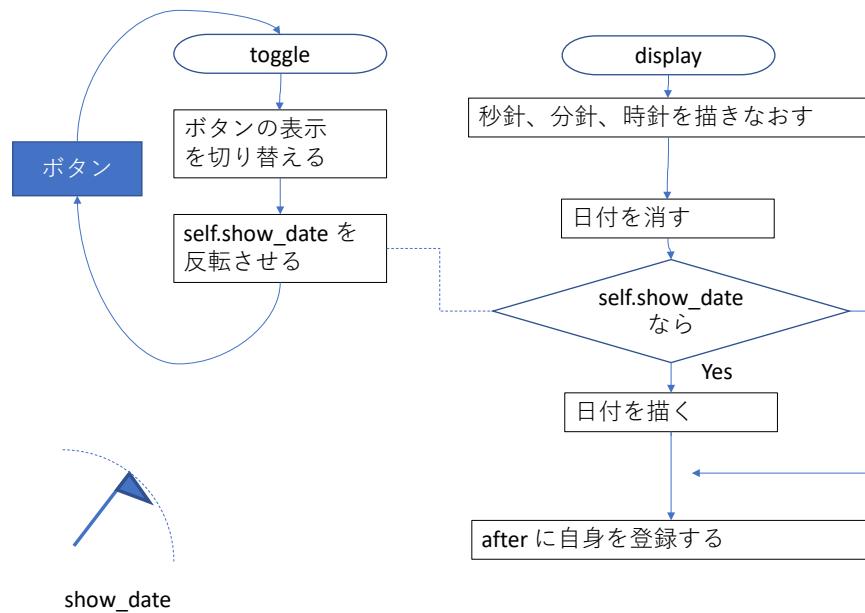


図 10-4 変数を介した動作の協調

# 11. クラス

---

## 11.1 本章の学習の目標

すでにタートルグラフィクスや `tkinter` でもクラス型オブジェクトの利用などをやってきましたが、ここではクラスについて以下を学びます。

- オブジェクト指向プログラミングの考え方を知る。
- クラスを定義して利用する。
- クラスで扱う変数について知る。

## 11.2 オブジェクト指向プログラミング

タートルグラフィクスで複数のタートルを扱うには以下のようなことを行いました。

- 必要なだけタートルを生成する。
- 個々のタートルに対してメソッドを呼び出す形で、動作を指示したり、状態を照会したりする。

個々のタートルは位置や向き、ペンの色やペンが降りているかどうか、などの状態を持っていました。

このタートルのように、それ自体が内部に状態などを持っていて、外からメソッドを呼び出すことで動作を指示できるようなものを「オブジェクト」と呼び、オブジェクトを使ってプログラミングする方法をオブジェクト指向プログラミングと呼びます。

`tkinter` のアナログ時計の例では、`tkinter` の `Frame` クラスを拡張する形でプログラミングを行いました。「クラス」は独自の状態やメソッドを持つオブジェクトを生成するための「型」の記述です。クラスから生成された個々のオブジェクトを「インスタンス」と呼びます。

簡単にまとめると

- (クラス型の) オブジェクトは独自の状態(変数)とメソッドを有するプログラムの要素、タートルのように命令できるロボットのような存在だと思う。
- クラスとはその型のオブジェクトがどのような変数とメソッドを持つかという記述。オブジェクトを生成するときの型となる。

- インスタンスはクラスを型に生成された個々のオブジェクト
- オブジェクト指向プログラミングはクラスの定義と生成されたインスタンスを用いてプログラムを作成する方法。ロボットを協調動作させるようにプログラムを書く考え方。

## 11.3 Python でのクラスの書き方、使い方

tkinter の例題と同様、二項演算をする CUI 型のプログラムを作成します。クラスを使って、第 1 項、第 2 項、演算結果、演算子を保持する変数と実際に演算を実行するメソッドをとりまとめます。

### 11.3.1 ソースコード

プログラム 11-1 CUI 型電卓プログラム (p11-1.py)

行	ソースコード	説明
1	class Dentaku():   2   def __init__(self):   3     self.first_term=0   4     self.second_term=0   5     self.result=0   6     self.operation= "+"	初期化メソッド
7		
8	def do_operation(self):   9   if self.operation=="+":  10     self.result=self.first_term+self.second_term  11   elif self.operation=="-":  12     self.result=self.first_term-self.second_term	計算を実行するメソッド
13		
14	#ここからメインプログラム	オブジェクトの生成
15	dentaku=Dentaku()	
16	while True:	
17	f=int(input("First term"))  18  dentaku.first_term=f  19  o=input("Operation")  20  dentaku.operation=o	
21	s=int(input("Second term"))  22  dentaku.second_term=s  23  dentaku.do_operation()  24  r=dentaku.result  25  print("Result",r)	

### 実行例

```
First term 1
Operation +
```

```

Second term 2
Result 3
First term 10
Operation -
Second term 5
Result 5
First term
KeyboardInterrupt
>>>

```

**Ctrl-C で中断**

Traceback (most recent call last):  
 File "M:/Documents/Python Scripts/class\_demo.py", line 17, in <module>  
 f = int(input("First term "))

### 11.3.2 プログラムの概要

- クラス定義のブロック（1～12行目），クラスの定義は以下のように行います

**class クラス名 ():**

メソッドなどの定義

- クラス名（この例では Dentaku）は変数と同様のルールで決めればいいのですが，慣習として先頭を大文字，残りを小文字にします。複数語での命名の場合，各語の先頭を大文字にし，単語間は詰めます。（例えば KansuuDentaku）
- メソッド **\_\_init\_\_(self)** の定義（2～6行）。Python では \_\_（アンダースコア 2つ）で始まるメソッドや変数などに特殊な役割を持たせることが多いのですが，  
**\_\_init\_\_()** はクラスのオブジェクトが生成される際に必ず実行されるメソッドで，オブジェクトの作り手の役割を担うのでコンストラクタとも呼ばれます（コラム「擬人化」も参照）。クラス内で使う変数の初期化などに使います。関数と異なり，クラスのメソッド定義では引数をかならず 1つ書かなければならず，通常 **self** という名前で与えます。呼び出しの際にはこの引数の値はシステムが自動的に与え，呼び出す際には第一引数は書く必要はありません。
- インスタンス変数と初期化（3～6行）。**\_\_init\_\_()** メソッド内で行っているのはクラスで使う変数の初期化です。
  - **self.** で始まる変数は「インスタンス変数」と呼ばれ，そのクラスのオブジェクトが生成されるごとにオブジェクト固有で，オブジェクトの中では永続的に使える変数です。
  - これに対し，**self.** を付けない変数は，関数と同様ローカル変数として扱われ，メソッドの処理が終わると捨てられます。

- メソッド `do_operation()` の定義 (8~12 行) : これは明示的に呼び出して使用するメソッドで、指定された演算を第一項と第二項を対象におこない、結果を書きこみます。引数 `self` が付されていること、処理の中で `self.` を付けてインスタンス変数を操作していることに留意してください。
- メインプログラム(14 行目以降)。端末から文字入力を受ける形で実行する電卓プログラムです。無限ループで記述しているので Ctrl-C で脱出します。
- クラス型オブジェクトの生成 (15 行目)。クラス型オブジェクトはクラス名を関数のように呼び出して、変数に代入することで行います。

**変数 = クラス名()**

- クラス型オブジェクトのインスタンス変数やメソッドの操作 (18~24 行)。クラス型オブジェクトの変数名に「`.`」でインスタンス変数名やメソッド名などを付けて呼び出します。`do_operation()` メソッドは定義では引数 `self` が必要ですが、呼び出しの際には不要であることを確認してください。

### 演習 11-1 Dentaku クラスの拡張

Dentaku クラスを乗算、除算も扱えるように拡張しなさい。ただし、除算は整数商でかまいません。

### 演習 11-2 複数のオブジェクトの生成と利用

Dentaku クラスのオブジェクトを複数生成して利用するプログラムを作成してください。もし足し算をしてくれるロボットをたくさん使えるなら何ができるか、と考えてください。例えば足し算をするロボットとそれを監視して検算（引き算）をするロボットとかはどうでしょうか。

### 演習 11-3 tkinter で作成した電卓プログラムでの Dentaku クラス利用

`tkinter` で作成した電卓プログラムについて、Dentaku クラスを利用するように改変しなさい

## 11.4 クラスの変数とアクセスの制限

先に Python のプログラムではプログラム全体に有効なグローバル変数と関数内で実行中に限り有効なローカル変数があることを述べましたが、クラスについてはこのほか、クラス変数とインスタンス変数について知っておく必要があります。

- クラス変数

- 生成：クラスの定義でメソッドの定義の外側で宣言します。
  - 動作：クラスで共通な変数として働きます。
  - アクセス：「クラス名.変数名」という形でクラス型オブジェクトを生成しなくとも参照できます。
- インスタンス変数
- 生成：メソッドの定義の中で `self.` を前につけて宣言します。
  - 動作：生成されたインスタンスごとに独立した変数として扱われます。そのインスタンスが使われている間は値を保持します。
  - アクセス：メソッドの定義の中では生成と同様 `self.` を前につけて参照します。
- 生成されたインスタンスを利用するプログラムからは、インスタンスを代入した変数（例えば `a`）の名前に「`.`」とインスタンス変数名を付けて参照します。

Python はあまり強力な変数の保護機能をもちません。クラス変数、インスタンス変数とも外から参照も書き換えも可能です。クラスの外からのアクセスを制限する方法として「アンダースコア 2つで始まる変数」の利用があります。このような変数はクラス内のメソッドからはアクセスできますが、クラス外から直接、操作はできません。

### プログラム 11-2 クラス変数とインスタンス変数 (p11-2.py)

行	ソースコード	説明
1	<code>#_クラスの練習</code>	
2	<code>class MyClass():</code>	クラス定義
3	<code>    #_以下はクラス変数</code>	
4	<code>    a = "マイクラス"</code>	<code>_b</code> は アクセス保 護される変 数
5	<code>    __b = 0</code>	
6	<code></code>	
7	<code>    #_以下は生成する際に呼ばれる関数,_mydata_の初期値を</code>	
8	<code>    #_引数で与える</code>	
9	<code>    def __init__(self,data):</code>	引数 <code>data</code> をとる初期 化メソッド
10	<code>        #__number_はインスタンスに与える通し番号</code>	
11	<code>        self.__number = MyClass.__b</code>	
12	<code>        self.mydata = data</code>	
13	<code>        print("MyClass Object is created, number:", self.__number)</code>	
14	<code>    #_クラス変数を 1 増やす</code>	
15	<code>    MyClass.__b += 1</code>	
16	<code></code>	
17	<code>    #_通し番号を表示するメソッド</code>	
18	<code>    def show_number(self):</code>	
19	<code>        print(self.__number)</code>	
20	<code></code>	

```

21  #
22  # ここからメインプログラム
23  #
24  if __name__ == "__main__":
25      print(" MyClass のクラス変数 a:", MyClass.a)
26
27      instance1 = MyClass(1)
28      instance2 = MyClass(10)
29
30      instance1.show_number()
31      instance2.show_number()
32
33      print("mydata of instance1:", instance1.mydata)
34      print("mydata of instance2:", instance2.mydata)
35      instance1.mydata += 1
36      instance2.mydata += 2
37      print("mydata of instance1:", instance1.mydata)
38      print("mydata of instance2:", instance2.mydata)

```

24行の指示によりモジュールとしてインポートされたときには実行しない

このプログラムを実行すると以下を得ます。クラス変数を用いてインスタンスに通し番号が付されていることや、インスタンス変数 mydata がインスタンスごとに独立であること、メインプログラムから直接アクセスできることが分かります。

```

 MyClass のクラス変数 a: マイクラス
 MyClass Object is created, number: 0
 MyClass Object is created, number: 1
 0
 1
 mydata of instance1: 1
 mydata of instance2: 10
 mydata of instance1: 2
 mydata of instance2: 12

```

またシェルで以下の操作をするとエラーが生じます。「\_\_」で始まるインスタンス変数が保護されていることが分かります。

```

>>> print(instance1.__number)
Traceback (most recent call last):
File "<pyshell#46>", line 1, in <module>
  print(instance1.__number)
AttributeError: 'MyClass' object has no attribute '__number'

```

なお、ソースコード中の

```
if __name__ == "__main__":
```

という表記はこのソースコードがメインプログラムとして実行された場合についてのみ実行するという指示です。ソースコードはモジュールとしてインポートすることも可能ですが、その場合はこの部分以降は実行されません。

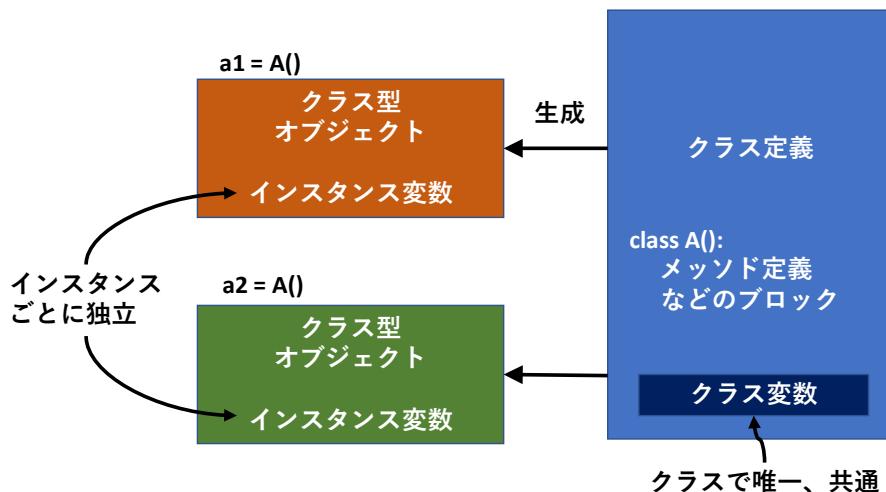


図 11-1 クラス変数とインスタンス変数

## 11.5 繙承

クラスを使ったプログラムの記述で重要なものに継承があります。本書では詳細は述べませんが、例えば `tkinter` の実装方法として紹介した例では `tkinter` の `Frame` クラスを継承したクラス `MyFrame` を定義して利用しました。これは `Frame` クラスとしての機能を継承したうえで、`Frame` 上のウィジェットの定義などを加える形で作られています。

## 11.6 インスタンスを起点にクラスを設計する

複雑なことを行うプログラムを作成する際にはクラスの利用は強力なツールになります。先に見たようにクラスはクラス型のオブジェクト（インスタンス）を生成する「型」という位置づけになります。しかしながら「型」からものごとを考えるのは難しいものです。

実際にクラスを設計する際には次のように具体的なインスタンスから考えるとよいでしょう。

- オブジェクトとしてまとめて取り扱いたいデータ（インスタンス変数の候補）と操作（メソッドの候補）を考える。
- このようなオブジェクトについてオブジェクトごとにクラスを考える。
- クラスをより広く使えるようにする
  - 複数のオブジェクト（クラス）で内容が同じなら、同じクラスでよい。

- 値の設定などが異なるだけならば、値をインスタンス変数にして、オブジェクトの生成の際に引数で与えることを考える。
- 共通のメソッドと個別のメソッドが混在する場合は、継承などを考える。

# 12. ファイル入出力

---

## 12.1 本章の学習の目標

1. Python で扱うファイルとしてテキストファイルについて知る
2. CSV 形式のファイルを通じて Python での計算結果を表計算ソフトで扱うこと を知る.
3. Python でのテキストファイルの読み書きについて知る.
4. ファイルの選択を支援する tkinter での filedialog について知る.

## 12.2 データを永続的に利用するには

これまでのプログラムではプログラム内の変数に設定されたデータはプログラムが稼働している間だけ保持され, プログラムが終了すると消されてしまいます. また, プログラムへの入出力は GUI にせよ CUI にせよ, 人が手で入力し, 結果は人が読む形で利用していました.

プログラムでデータを永続的に利用するには, データをプログラムの外側で保存できる形で書き出したり, 読み込んだりする必要があります. その候補としては

- コンピュータ上のファイル
- コンピュータ上のデータベース
- ネットワーク上のサービス

などがありますが, ここでは基礎となるコンピュータ上のファイルの操作について学びます.

## 12.3 ファイルについて

### 12.3.1 ファイルパス

コンピュータでのファイル扱いは Windows, macOS, linux などのオペレーティングシステムが管理しています. これらの OS では階層的なフォルダ（ディレクト

リ) <sup>1</sup>構造、すなわち、フォルダの中にフォルダを置ける構造をとっています。ファイルの所在はフォルダ構成上の位置で特定するようになっています。ファイルがどこにあるかを記述した文字列は「ファイルパス(file path)」と呼ばれ、「階層的なフォルダ構成の表記」と「ファイル名」が結合した形をとります。

具体的な例を挙げれば Windows では

**M:¥documents¥python scripts¥ex1.py**

といったものがファイルパスです。フォントによっては

**M:\documents\python scripts\ex1.py**

と表示されます。

ここで

**M:** ドライブ名 (ディスク装置やファイルサーバに対応します)

**¥document¥python\_scripts:** フォルダのパス

**ex1.py:** ファイル名

**.py:** ファイル名の . 以降を特にファイルの種類を表す「拡張子」と言います。

なお Windows ではフォルダの区切りを表す文字として「¥」(日本語環境では¥で表示され、それ以外では逆スラッシュ (\)) 使われます。

ドライブ名からフォルダをすべて表記したものを「フルパス」と言います。フルパスならば、そのコンピュータ上のファイルを唯一に特定できます。

これ以外に、「現在、作業中のフォルダ (ディレクトリ, current working directory, cwd)」というものが設定されていて、そこからの差分だけを記述したものを「相対パス」と言います。例えば、作業中のフォルダが

**M:¥documents¥python scripts**

であれば相対パス表記

**ex1.py**

は

**M:¥documents¥python scripts¥ex1.py**

を意味します。

### 12.3.2 Raw String の利用

ファイルのパスを直接、ソースコードの中に書く場合は Python が '¥' を特殊な文字として解釈することに注意しなければなりません。例えば '¥n' は改行を意味し

<sup>1</sup> ファイルをまとめて整理する仕掛けとして Windows では「フォルダ」を使いますが、その前身である MS-DOS や unix ではディレクトリ (directory, 住所録) をいう言葉が用いられています。厳密には少し異なるのですが、ここではフォルダとディレクトリは同じものと考えてください。

ます。ファイルパスにはフォルダの区切りに「¥」を用います。これを正しく扱うには以下のように「¥」を2つづつ書くか

```
filepath = "M:¥¥documents¥¥python scripts¥¥p3-1.py"
```

'r'という文字を前置して、そのままの文字列(raw string)として扱う必要があります。コラム「逃げる」も参照してください。

```
filepath = r"M:¥documents¥python scripts¥p3-1.py"
```

### 12.3.3 テキストファイル

テキストファイルとは「文字コード（と改行などの記号）で書かれたファイル」で、エディタなどを使えば人にも読み書きできるファイルの形式を指します。例えばPythonのソースコードや電子メールのメッセージはテキストファイルです。

これに対してコンピュータの内部形式のデータで構成されたファイルを「バイナリファイル」と呼びます。「バイナリ」とは「2進数の」という意味です。バイナリファイルはコンピュータ内部の形式をそのままファイルに書き出したもので、数値などは精度を失わない、データ量が少ないなどのメリットがありますが、ファイル内容の記述などがなければ何が書かれているのかは分かりません。

本章ではPythonでテキストファイルを読み書きすることを学びます。

### 12.3.4 CSV 形式

Pythonで作成したプログラムを他のツールと連携して利用できるとあまり手間をかけずに応用範囲が広がります。このためには簡単にデータを扱える形式としてCSV(Comma Separated Value)形式があります。これはテキストファイルの一形式で各行が

データ1, データ2, データ3

などのようにデータとデータの間をカンマ「,」で区切った形式です。この形式のファイルのファイル名に.csvという拡張子を付けておけばExcelなどの表計算ソフトで読み込むことができ、グラフ作成などが簡単に行えます。

CSV形式のデータの出力は比較的簡単です。他方で読み込みは「カンマや改行を含む文字列の扱い」など面倒な問題もあり、データの内容によってはライブラリの

活用<sup>1</sup>などを考えるほうがよいでしょう。

### 12.3.5 文字コードの問題

日本語の文字コードは歴史的な経緯で、複数併存しており、しかも扱う OS によって使われる文字コードが異なっています。例えば日本語のファイル名に使われる文字コードは以下のようになっています。

- Mac, Linux: Unicode
- Windows: Shift-JIS

テキストファイルでは上記の文字コードの違いに加え改行を表すコードも違います。

- Python 3 は内部では Unicode の表現方法の 1 つである UTF-8 で扱います。IDLE で作成した Python のプログラム（スクリプト）も UTF-8 でコード化されて保存されています。1 つの OS 内で Python を実行する場合は OS による差異を Python が適宜、調整してくれますのであまり気にしなくてよいのですが、異なる OS でプログラムを動作させる場合には注意が必要です。

### 12.3.6 エラー処理

ファイルの入出力ではエラー処理は極めて重要です。これはファイルやファイルシステム、読み込むデータの内容はプログラムでは統制できないためです。ファイルを開こうとしたら、ファイルやフォルダが存在しなかったり、書き込み権限がなかったり、書いている途中でディスク領域が不足したりなど、さまざまなことが生じる可能性があることを意識しなければなりません。

## 12.4 まずは動かしてみよう

### 12.4.1 ソースコード

プログラム 12-1 ファイル入出力の例題 (p12-1.py)

行	ソースコード	説明
1	# 今のワーキングディレクトリ（作業中のフォルダ）	
2	# を調べるために os モジュールを import します	

<sup>1</sup> Python では csv を扱うライブラリとして csv モジュールがあります。後の章では pandas で csv ファイルを読み込む例を紹介します。

```

3 import os
4 # 今のワーキングディレクトリを得て画面に表示します
5 print(os.getcwd())
6 # 「日本語ファイル.txt」という名称のファイルを作成し、内容を書
7 f = open('日本語ファイル.txt', 'w')
8 f.write('日本語\n日本語\n日本語\n')
9 f.close()
10 # 「日本語ファイル.txt」を読み込み用に「open」して、その内容を表
11 f = open('日本語ファイル.txt', 'r')
12 s = f.read()
13 f.close()
14 print(s)

```

Windows では \ は  
フォントによって  
¥ と表示される

## 12.4.2 プログラムのポイント

- ・ 今の作業フォルダ（カレントワーキングディレクトリ）を知る(8行目)
- ・ 「日本語ファイル.txt」という名前のファイルを書き込み用(w)に開き、以後 f という変数で扱う。相対パスとして表記されているので作業フォルダにこの名前で作成される. (10 行)
- ・ ファイルへの文字列の書き出し (11 行) 「\n」はこの 2 文字で「改行」を意味します。
- ・ 書き出し用のファイルを閉じる(12 行)
- ・ 同名のファイルを読み出し(r)用に開く. (15 行)
- ・ ファイルの内容を変数 s にすべて読み出す. (16 行)
- ・ ファイルを閉じる(17 行)
- ・ 読み出したデータ（テキスト）の出力(18 行)

## 12.5 Python でのファイルの読み書き

### 12.5.1 open 関数の利用

以下の手順でファイルを操作します。

1. open 関数でファイルを開き、返り値でファイルオブジェクトを得る。

`file = open(ファイル名, モード)`

モードは読む "r" , 書く "w" など。上の例では返り値を変数 file に代入しています。

なお、Python では特に指定しなければテキストファイルの文字コードは稼働

している OS の標準の文字コードを想定します、明示的に文字コードを指定するには例えば以下のように `encoding` 引数を設定します。

```
file = open(ファイル名, モード, encoding="utf-8")
```

ファイルを開くのに失敗した場合は `IOError` という例外が発生します。

## 2. ファイルオブジェクトへの読み書き

(ア) ファイルオブジェクトから `read()` メソッドで読みこむ

```
s = file.read()
```

上の例ではテキストファイルから全体を文字列として読んで変数 `s` に代入しています。

(イ) ファイルオブジェクトに `write()` メソッドで書き込む

```
file.write(s)
```

上の例では文字列型のデータ `s` をファイルに書き込みます。

ファイルを閉じるまで同様にして追記可能です。

## 3. ファイルを閉じる

```
file.close()
```

注意：`open` は組み込み関数、`read`, `write`, `close` はファイルオブジェクトのメソッド

上の例ではファイルの内容をすべて一括して読み込みました。1行だけ読み込むには `readline()` メソッドを使います。また、以下のように `for` 文でファイルの内容を1行ずつ処理することも可能です<sup>1</sup>。

```
file = open("ファイル名", "r")
```

```
for line in file:
```

一行ずつ処理するブロック

## 12.5.2 `with` 文の利用—`close()` の自動化

`open()` 関数で開いたファイルは `close()` メソッドで閉じる必要がありますが、以下の理由で `close()` が行われない場合があります。

- 単純に `close()` メソッドを呼び忘れている。
- エラーなどで `close()` メソッドを書いている箇所が実行されない。

---

<sup>1</sup> Python の `for` 文は `range()` 関数、文字列（1文字ずつ）、リスト（要素ごと）など様々な対象に適用できますが、これらは「繰り返し処理が可能な対象」としてイテレータという性格を与えられているからです。ファイルオブジェクトも「一行ずつ」という形でイテレータとして使えます。

これらを避けるために Python では `with` 文が用意されており、`with` 文で開いたファイルはブロック終了後に自動的に閉じられます。

`with open(ファイル名など open 関数の引数) as ファイルオブジェクト用変数:`  
     ファイルを操作するブロック

## 12.6 例題 1 波の近似

### 12.6.1 例題のポイント

- ・ ファイルパスを正確に端末から入力するのは面倒なので `tkinter` (の `filedialog` だけ) を使います。
- ・ 計算結果を `csv` 形式で出力して、表計算ソフトと連動します。
- ・ 例題として周期関数を三角関数の和で表現する例を用います。

### 12.6.2 周期関数の三角関数の和での近似

周期関数（ある周期で値が繰り返す関数）はその周期の整数倍の正弦関数( $\sin$ )と余弦関数( $\cos$ )の和で近似できることが知られています。のこぎり波（鋸歯状波、鋸の歯のような波）は以下のように近似できることが知られています（コラム「三角関数」も参照してください）。

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3} + \frac{\sin(4x)}{4} \dots$$

下の図は第 1 ~ 第 5 項までの和をプロットしたもの。

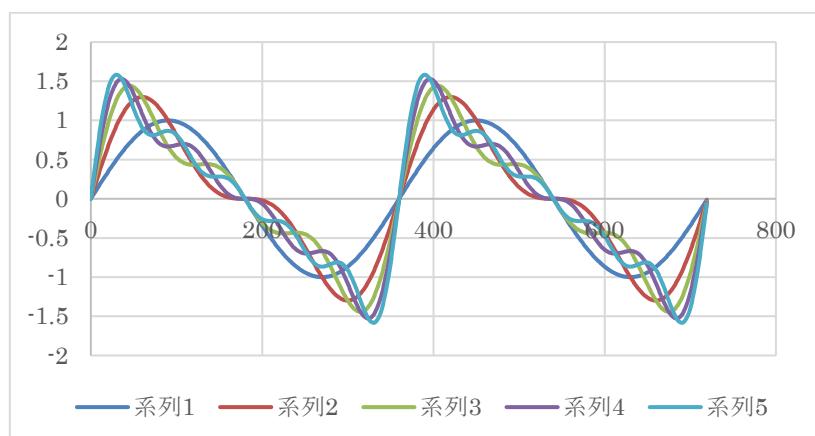
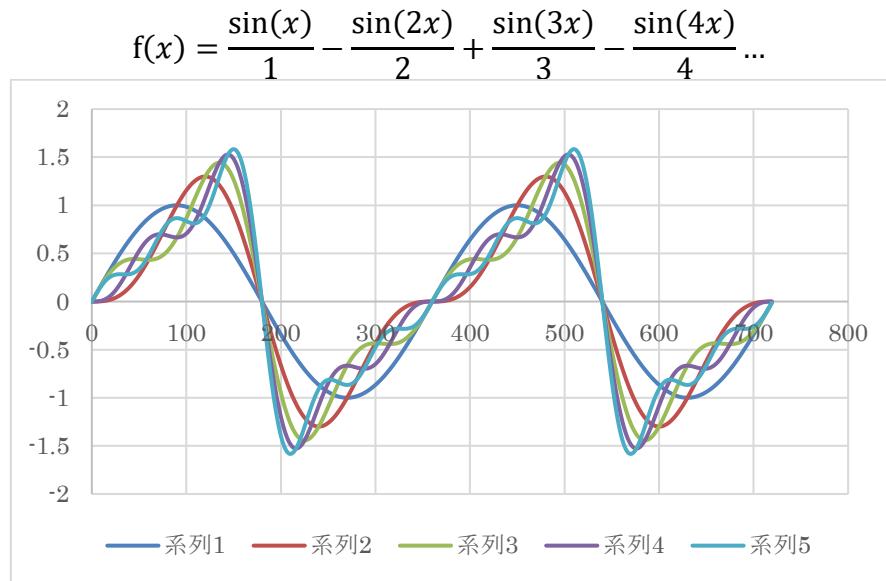


図 12-1 のこぎり波の三角関数の和での近似

上記は簡単のため、各項の符号を揃えていますが、原点を傾き正で通るのこぎり波については以下のように交互に符号が変わります。



また、振幅が 1（最大、最小値が  $\pm 1$ ）ののこぎり波は全体に係数  $(2/\pi)$  がかかります。

### 12.6.3 ソースコード

プログラム 12-2 のこぎり波の三角関数の和での近似 (p12-2.py)

行	ソースコード	説明
1	import tkinter as tk	
2	import tkinter.filedialog	filedialog もインポート
3	import math	
4	#	
5	#tkinter の filedialog だけを利用する例	
6	#	
7	#root ウィンドウは withdraw() メソッドを読んで隠す	
8	root=tk.Tk()	
9	root.withdraw()	
10	#	
11	#書き出し用の filedialog を読んでファイル名を得る	ファイル名をダイアログで得て戻ります。
12	#	
13	filename=tkinter.filedialog.asksaveasfilename()	
14	#	
15	#tkinter は終了する	tkinter はもう使わないので終了します。
16	#	
17	root.destroy()	
18	#	
19	#ファイル名がもらえなければ終了	

<pre> 20  # 21  if filename: 22      pass 23  else: 24      print("No file specified") 25      exit() 26  # 27  # 正弦波の重ね合わせで鋸波を近似する 28  # 29  # w = sin(t) + sin(2t)/2 + sin(3t)/3 + sin(4t)/4... 30  # 31  # 2 周期分、全体は 1000 ステップで、高調波は 5 番目まで 32  # 33  cycles = 2 34  steps = 1000 35  harmonics = 5 36  # ファイルが開けないときのエラー対応 37  try: 38      # ファイルを開く 39      with open(filename, 'w') as file: 40          for i in range(steps): 41              angle_in_degree = 360 * cycles * i / steps 42              angle = math.radians(angle_in_degree) 43              s = str(angle_in_degree) 44              w = 0 45              for j in range(1, harmonics + 1): 46                  w += math.sin(angle * j) / j 47              s = s + "," + str(w) 48              # print(s) 49              file.write(s + "\n") 50          print("Writing to file " + filename + " is finished") 51      except IOError: 52          print("Unable to open file") </pre>	<p>pass は特に処理をしない命令</p> <p>角度を文字列に</p> <p>和 w を「,」を前において s に追加 s をファイルに追記、改行「\n」を加えて書き出し</p>
--	--

## 12.6.4 プログラムのポイント

### 1) tkinter の filedialog の利用（1～25 行）

- Windows のアプリケーションなどでファイルを開いたり、ファイルに保存したりする際に別ウィンドウでファイルを探したり、指定したりできます。tkinter ではこのための仕組みとして filedialog が用意されています。このプログラムでは filedialog の機能だけを使うので tkinter のメインウィンドウは 8 行目で作成しますが、使わないので 9 行目で見えなくしています。また mainloop() メソッドは呼んでいないことに注意してください。

- `filedialog` では利用目的によりいくつかの形式があるのですが、ここでは「名前をつけて保存」用の `asksaveasfilename()` メソッドを 13 行目で呼び出し、返り値で得られるファイル名（パス名）を `filename` で得ます。
- `filedialog` から戻れば `tkinter` は不要なので、`root.destroy()` で `tkinter` を終了しています。
- キャンセルなどの操作の場合、`filename` には何も入りませんので、`if` 文の `false` の場合にプログラムを終了しています。
- 読み出し用の `filedialog` の利用については次の例題を参照してください。

## 2) 計算とファイル出力（33 ~ 52 行）

- この部分が三角関数の重ね合わせを計算して CSV 形式で出力している部分です。
- ファイルの取り扱いではファイルを開くことができないなどエラーに対応する必要があり 37 行目で `try` 文でファイル操作するブロックを扱っています。対応するエラー処理は 51, 52 行です。
- 39 行目で `with` 文でファイルを開いています。`filedialog` で得たファイル名 `filename` のファイルを開き、開いたファイルは変数 `file` で扱っています。
- 計算し、出力する 1 行の内容は

角度、第 1 項、第 2 項までの和、第 3 項までの和、第 4 項までの和、第 5 項までの和です。和は変数 `w` に計算してゆき、1 行の内容は変数 `s` に文字列として書き加えています。CSV 形式にするため、`w` の値を文字列に変換して `s` に加える際に

```
s = s+", "+ str(w)
```

と `","` を間に挟んでいます。カンマのあとに<sup>1</sup>スペースを入れているのは出来たファイルを読みやすくするためです。

- `for` 文で第 5 項までの計算が終わったあと、48, 49 行目でファイルに出力しています。

```
#     print(s)
file.write(s+"\n")
```

48 行名はコメントになっていますが、Python Shell で結果を確認したい場合は `#` を削除してください。49 行目では `file.write()` でファイルに書き出していますが、1 行の文字列 `s` に「改行」を加えるために「\n」が付加されています。

---

<sup>1</sup> 実行できるプログラムの一部をコメントにして、実行を抑止する方法は「コメントアウト」と呼ばれ、プログラムの動作確認などでしばしば用いられます。

Windows ユーザは利用するフォントによっては「\」ではなく円記号「¥」を入力してください。

### 演習 12-1 矩形波（方形波）の近似

矩形波（方形波）（±1 の値を交互にとる周期関数）は以下のように三角関数で近似できます。<sup>1</sup>

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \frac{\sin(7x)}{7} \dots$$

例題と同様の方法で方形波を三角関数での近似を計算し、csv ファイルに出力した後、表計算ソフトでグラフを作成してください。

### 演習 12-2 例題 1 のリストを使った実装

例題 1 のプログラムでは計算結果は逐次、文字列として結合し、1 行ごとにファイルに書き出しています。これを以下のように計算と出力を分離した形で再実装してください。

- 計算結果はリストを使ってリスト上に書き込む。
- 計算の終了後、そのリストを参照する形で例題 1 と同じ形式の CSV ファイルを書き出す。

なお、リストの構成法として、以下の 2 通りの考え方があります。どちらの実装法でも構いません。

---

<sup>1</sup> 振幅 1 の矩形波については全体に係数  $4/\pi$  がかかります。

```
[ [ 時刻(0), 第 1 項(0), 第2項までの和(0), 第3項までの和(0), 第4項までの和(0)] ,  
  [ 時刻(1), 第 1 項(1), 第2項までの和(1), 第3項までの和(1), 第4項までの和(1)] ,  
  [ 時刻(2), 第 1 項(2), 第2項までの和(2), 第3項までの和(2), 第4項までの和(2)] ,  
  [ 時刻(3), 第 1 項(3), 第2項までの和(3), 第3項までの和(3), 第4項までの和(3)] ]
```

図 12-3 「各時刻のデータのリスト」のリストとして扱う

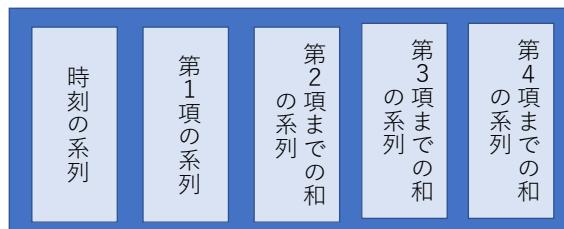


図 12-4 「各系列のリスト」のリストとして扱う

## 12.7 例題 2 テキストエディタ

tkinter を用いれば簡単なテキストエディタを作ることができます。tkinter の機能としてはこれまでに学んだもののほか、メッセージ表示用のダイアログである messagebox, filedialog のファイル読み込み用のメソッド、tkinter の Menu ウィジエット、Text ウィジェットなどを使っています。ジオメトリマネージャは構成が单纯なため grid ではなく pack を使っています。

また、ファイルの漢字コードは特に指定していないので Python では OS ごとの標準的コードを想定します。Windows では Shift-JIS コード(cp932)が使われているとして扱われます。

## プログラム 12-3 tkinter を用いた簡単なテキストエディタ(p12-3.py)

行	ソースコード
1	import tkinter as tk
2	import tkinter.messagebox
3	import tkinter.filedialog
4	# messagebox, filedialog は明示的なインポートが必要
5	#
6	# tk.Frame を継承した MyFrame というクラスを作り
7	# その中でウィジェットやコールバック関数(メソッド)を
8	# 設定する。tkinter をつかう定番
9	#
10	class MyFrame(tk.Frame):
11	# __init__ はクラスオブジェクトを作る際の初期化メソッド
12	def __init__(self, master=None):
13	super().__init__(master)
14	self.master.title('Simple Editor')
15	
16	# メニューを作る。menubar->filemenu->Open, Save as, Exit
17	menubar = tk.Menu(self)
18	filemenu = tk.Menu(menubar, tearoff=0)
19	filemenu.add_command(label="Open", command=self.openfile)
20	filemenu.add_command(label="Save as...", command=self.saveas)
21	filemenu.add_command(label="Exit", command=self.master.destroy)
22	menubar.add_cascade(label="File", menu=filemenu)
23	self.master.config(menu=menubar)
24	
25	# 編集用 Text ウィジェットをクラスの変数 editbox としてつくる
26	self.editbox = tk.Text(self)
27	self.editbox.pack()
28	
29	# ファイルを開くメソッド。_ 関数とちがい self という引数が必要
30	def openfile(self):
31	# filedialog でファイル名を得る
32	filename = tkinter.filedialog.askopenfilename()
33	# filename が空でなければ処理
34	if filename:
35	tkinter.messagebox.showinfo("Filename", "Open: "+filename)
36	# with 文で file という変数でファイルを開く
37	with open(filename, 'r') as file:
38	text = file.read()
39	# Text ウィジェット editbox にファイル内容を設定
40	self.editbox.delete('1.0', tk.END)
41	self.editbox.insert('1.0', text)
42	else:
43	tkinter.messagebox.showinfo("Filename", "Canceled")
44	
45	# ファイルに保存するメソッド
46	def saveas(self):
47	# with 文で file という変数でファイルを開く
48	filename = tkinter.filedialog.asksaveasfilename()
49	if filename:

```
50     with open(filename,'w') as file:
51         text = file.write(self.editbox.get('1.0',tk.END))
52         tkinter.messagebox.showinfo("Filename","Saved AS:"+filename)
53     else:
54         tkinter.messagebox.showinfo("Filename","Canceled")
55
56 # ここからメインプログラム
57 root = tk.Tk()
58 f = MyFrame(root)
59 f.pack()
60 f.mainloop()
```



# 13. 三目並べで学ぶプログラム開発

---

## 13.1 本章の学習の目標

この章では三目並べを例に課題を与えられてプログラムを開発することを学びます。

1. 三目並べをプレイすることを分析し、プログラムで表現する必要のある事項を洗い出します。
2. プログラムのテストに備えて棋譜を準備します。
3. プログラムを構成するデータや関数を小さなものから順に作成します。
4. 全体を組み上げて三目並べのプログラムを完成させます。

## 13.2 プログラムを開発するということ

これまでのいくつかのプログラムを構成してきましたが、何か具体的な課題を与えられてプログラムを作ることは、それ自体が初学者には難しいことのようです。プログラミング言語のさまざまな要素を使えるようになると、まとまったプログラムを1から開発することは異なる能力だからです。例えると、金槌や鋸を使えるからと言って家を建てることができる訳ではありません、家を建てるには、家とはどのような構成になっていて、どのような順番で設計し、施工する必要があるかを知っていなければならないからです。プログラムも同様です。

## 13.3 設計手順—コンピュータを使う前にすること

初心者の方で、プログラムを作ろうとしてコンピュータを開き、そこで動けなくなる人を良く見かけます。なぜ動けなくなるのでしょうか？

プログラムの設計・作成手順は以下のようになります。最初からコンピュータが必要な訳ではないのにコンピュータに向かってしまうから動けなくなるのです。

- コンピュータを使う前にすること
  - 実現したいことを言葉で表現する
  - プログラムとして作るべきものを特定する

- ✧ 変数として表現すべきこと
- ✧ 変数のとる値として表現すべきこと
- ✧ 手順（関数）として表現すべきこと
- ✧ 人とのやりとりとして表現すべきこと
- プログラムとして作成する順序を決める
- テストの方法を決める
- ここからコンピュータでの作業
  - 他に依存しない部分からプログラム（関数）を作る
  - 作った関数をテストする（単体テスト）
  - 全体をテストする（結合テスト）

## 13.4 三目並べを例にしたプログラムの設計

### 13.4.1 三目並べ (tic-tac-toe)

三目並べのルールとゲームの進行を言葉で表現してみてください。

○	×	
	○	
○		×

### 13.4.2 文章の分析

三目並べのルールとゲームの進行を表現した文章から品詞（名詞, ~だ, ~である, という表現, 動詞）に着目して文章を分析します<sup>1</sup>。以下のような事項が得られます。

- 特定の状態をとる事項（名詞）：変数の候補です
  - $3 \times 3$  マスの盤面, 手番
- 事項の状態（~だ, ~である）：変数の取り得る値の候補です
  - 各マスの状態（空, ○（先手）, ×（後手））
  - どちらの手番か
- 状態を調べる動作（関数の候補です）

<sup>1</sup> こういう作業があるので、プログラミングには文科系的素養が大事なんです。

- どちらの手番か
- マスの状態
- 先手勝ち, 後手勝ち, 引き分け
- 状態を変える動作（関数の候補です）
  - マスに○や×を置く
  - 手番を入れ替える

### 13.4.3 棋譜の作成—テストの準備として

プログラムを作る前にテスト用の棋譜をいくつか作成しておきましょう。すべての場合をつくすことは難しいですが、以下のようなことを考えて準備します。

- 先手勝ち, 後手勝ち, 引き分けを含むこと
- 勝ち方のパターン（縦（3通り）, 横（3通り）, 斜め（2通り））を含むこと
- テ스트ケースを先に用意することは「**テストファースト**」と呼ばれ、プログラムを作り始めること「**コーディングファースト**」より以下のような点で効果があるとされています。
- 後からだとテストケースの作成に手を抜く
- プログラムの作成中、いつでもテストできる
- テストすることをコーディングで意識できる

手番	row	column	row	column	row	column
1 ○	0	0	0	0	0	1
2 ×	1	1	1	0	0	0
3 ○	1	0	1	1	2	1
4 ×	2	0	2	2	1	1
5 ○	0	2	0	2	2	2
6 ×	0	1	0	1	2	0
7 ○	2	1	2	0	1	0
8 ×	2	2			0	2
9 ○	1	2				
結果	引き分け		先手勝ち		後手勝ち	
	0	1	2	0	1	2
	0 1○	6×	5○	0 1○	6×	5○
	1 3○	2×	9○	1 2×	3○	
	2 4×	7○	8×	2 7○	4×	

図 13-1 三目並べ、棋譜の例

## 13.4.4 変数の設計

### 1) 盤面

- $3 \times 3$  の盤面を 2 重のリスト（要素は整数）で表すことにします。

```
board = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

- 初期状態はすべて 0（空き）
- 値の意味は 0 空き, 1 先手 (○), 2 後手 (×) とします。
- このために定数（Python では値の変更を禁じる方法はありません）を定義します。定数であることを意識するため大文字を使います。手番や結果への利用も考慮して以下のようにします。

```
OPEN = 0
FIRST = 1
SECOND = 2
DRAW = 3
```

### 2) 手番

- どちらの手番(turn)かを整数変数で表します。初期値は FIRST(先手)
- turn = FIRST
- 値は先の定数 (FIRST, SECOND) を利用し FIRST: 先手と SECOND: 後手

### 3) 棋譜

- 手番は「先手→後手→先手…」と決まっているので、打った場所の行、列のリストで表現します。
- リストの最後に結果（未定、先手勝ち、後手勝ち、引き分け）を入れることにします<sup>1</sup>。
- 棋譜 = [[初手の行, 列], [二手目の行, 列], ..., [最終手の行, 列], [結果]]
- 行、列はそれぞれ 0, 1, 2 の整数、結果は 0, 1, 2, 3 の整数

---

<sup>1</sup> 棋譜を 1 つのリストで表現したため、リストの要素は「打った場所」と「結果」という異なる意味を持ってしまいます。あまり分かりやすい実装とは言えません。

### 13.4.5 盤面と手番に関する関数

状態の「操作」，状態の「検査」，画面に「表示」する，初期化（操作のひとつ）するなどの関数を作ります。

盤面や手番などをグローバル変数として共有するので，関数内で関数外の変数の値を変えるためには `global` 宣言が必要です。以下のような関数が必要でしょう。

#### 1) 手番について

- 操作：手番を初期化する
- 操作：手番を交代する
- 表示：手番を表示する（ための文字列を生成する）

#### 2) 盤面について

- 操作：盤面を初期化する
- 操作：盤面の指定されたマスを指定された手番にマークする
- 検査：盤面の個々のマスの状態を知る
- 検査：盤面がどちらの手番の勝ちであるかを知る
- 検査：盤面がすべて埋まっているかを知る
- 表示：盤面全体を出力する（ための文字列を生成する）

#### 3) 棋譜について

- 操作：棋譜を用いて対戦を再生する

#### 4) 勝敗判定のアルゴリズム

日頃，皆さんのが実際にプレーしているゲームですが，明示的に勝敗判定の方法を書き出すと以下のようになります。

- ある行・列方向にどちらかの手番（以下，`t` とします）の勝ちを判定する。
  - もし注目する方向上の 3 つの位置のコマがすべて `t` ならば勝ち，
  - そうでなければ勝ちではない
- ある方向（横，縦，対角，逆対角）に勝ちかどうかを判定する
  - 横，あるいは縦方向なら
    - ❖ もし第 0 行目（第 0 列目）が勝ちならば勝ち
    - ❖ そうでなければもし第 1 行目（第 1 列目）が勝ちならば勝ち
    - ❖ そうでなければもし第 2 行目（第 2 列目）が勝ちならば勝ち

- ◆ そうでなければ勝ちではない
- 対角あるいは逆対角方向なら、その方向に勝ちならば勝ち
- 上記を使った勝ち判定

1. もし横方向に勝ちならば勝ち.
2. そうでなければもし縦方向に勝ちならば勝ち.
3. そうでなければもし対角方向に勝ちならば勝ち.
4. そうでなければもし逆対角方向に勝ちならば勝ち.
5. そうでなければ t の勝ちではない

なお、逐次勝敗を判定している盤面では生じませんがランダムに生成した盤面では先手、後手ともに勝っているという状況があり得ます。

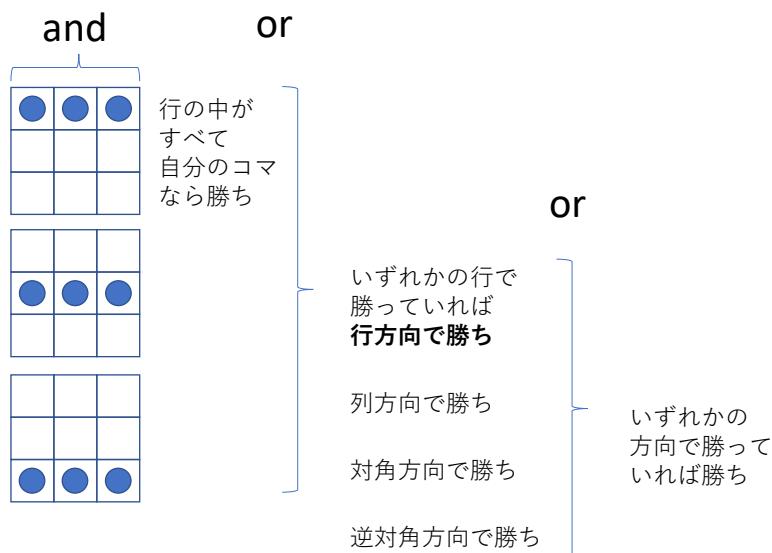


図 13-2 着目する手番の勝ち判定

### ● 勝敗を判定する

上の手続きを（関数）を用いて三目並べでは以下のように勝敗を判定できます。

1. 先手が勝っているかどうかを調べ、もし先手が勝っていれば「先手勝ち」
2. そうでなければ後手が勝っているかどうかをしらべ、もし後手が勝っていれば「後手勝ち」
3. そうでなければ、もし盤面に空きがあればまだ「未決着」
4. そうでなければ（盤面に空きがない）ならば「引き分け」

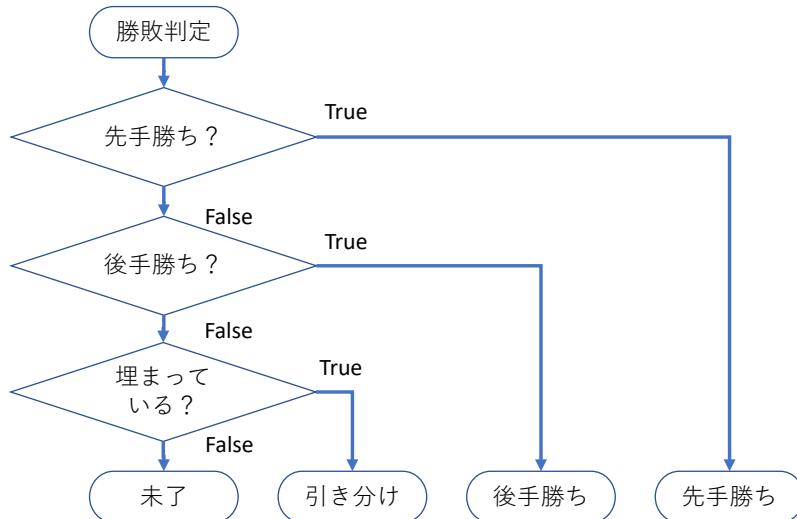


図 13-3 勝敗判定

### 13.4.6 複雑な条件判断の書き方

4章でも少し説明しましたが、実際に三目並べの勝敗の判定は結構、複雑です。例えば3つの条件 `is_A`, `is_B`, `is_C` (`True`, `False` の値をとる変数や、関数を考えてください)があるとして、これらについて、すべてが `True` の場合の判断をする関数の判断部分は

```
return is_A and is_B and is_C;
```

と書いてもいいですし、`if` 文を入れ子にして

```
if is_A:
    if is_B:
        if is_C:
            return True
    return False
return False
```

とも書けます。また、

```
if not is_A:
    return False
if not is_B:
    return False
if not is_C:
    return False
return True
```

と書くこともできます。最後のやり方はリストを使うと

```
conditions = [is_A, is_B, is_C]
for c in conditions:
    if not c:
        return False
return True
```

と書くこともできます。最後のやり方は調べる条件が多くなっても `for` 文の利用で判断部分をコンパクトに書けます。

### 演習 13-1 複雑な条件判定の書き方

`is_A`, `is_B`, `is_C` のいずれかが `True` の場合に `True` と判定する関数の書き方を上の例に倣って考えてみてください。

## 13.4.7 ゲームの進行

メインプログラムの流れを考えます。入出力は Python Shell 上での文字での入出力を考えます。先に定めた関数を使って容易に実装できるかを確認します。

- ゲームを初期化する
- 盤面を表示する
- 以下、勝ち負け、引き分けまで繰り返す
  - 正しい入力が得られるまで手番側の入力を促進する
    - ◆ 手番側の入力を得る
  - 盤面を更新する
  - 盤面を表示する
  - 勝ち負け、引き分けを判定する
    - ◆ ゲーム終了なら結果を示して脱出
  - 手番を交代する

## 13.5 プログラムの実装

### 1) ソースコードの構成

設計が完了したらプログラムの実装を始めます。Python のソースコードは以下のような構成にすることを確認しておきます。

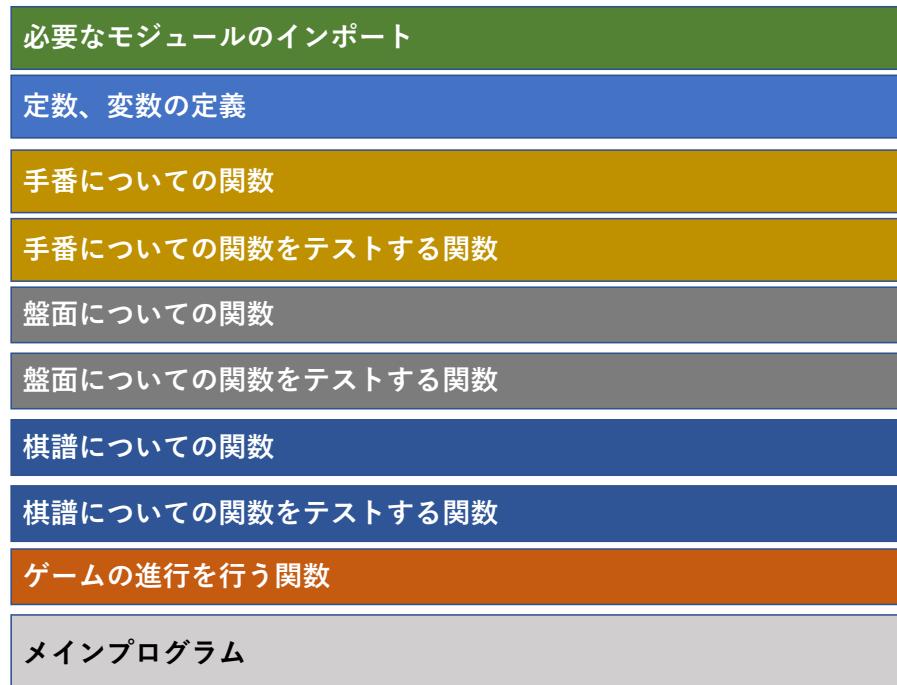


図 13-4 ソースコードの全体構成

## 2) 実装例 (`tic_tac_toe.py`)

プログラムの実装例を以下に示します。実装の方針や記法についてポイントを説明します。

- プログラムは図 29 に示した順に実装されています。
- プログラムは全体で 500 行近くありますが、背景を黄色にした部分はテスト用の関数です。
- 関数等に `docstring` を付しています。複数行にわたる `docstring` は "" (シングルクオート 3 つ)で開始し、"" で終わるという記法を用いています。
- 特にインポートしなければならないモジュールはありません。
- 表示するための関数は、「画面に表示する」のではなく、`print()` 関数に渡せる文字列を生成する形をとっています。
- メインプログラムは単に「三目並べ」と表示するだけです。すべての関数が読み込まれた状態になっていますので、Python Shell でテスト用の関数や実際のプレイ用の関数を呼び出すことができます。

## プログラム 13-1 三目並べのプログラム、実装例（その 1 グローバル変数）

行	ソースコード
1	#
2	#_三目並べ
3	#
4	#_特にインポートするモジュールはありません
5	#
6	#_定数の定義
7	#
8	#
9	#_play()の中で棋譜を作成する（要完成）
10	#
11	'三目並べのプログラムです'
12	OPEN_=0
13	FIRST_=1
14	SECOND_=2
15	DRAW_=3
16	#
17	#_恒常的な変数
18	#
19	turn_=1
20	board_= [[0,0,0],[0,0,0],[0,0,0]]
21	#
22	#_テスト用の棋譜
23	#
24	log1_= [[0,0],[1,1],[1,0],[2,0],[0,2],[0,1],[2,1],[2,2],[1,2],[ EVEN ]]
25	log2_= [[0,0],[1,0],[1,1],[2,2],[0,1],[2,0],[ FIRST ]]
26	log3_= [[0,1],[0,0],[2,1],[1,1],[2,2],[2,0],[1,0],[0,2],[ SECOND ]]

## プログラム 13-2 三目並べのプログラム、実装例（その 2 手番関連の関数）

```
27  #
28  # 手番関連の関数
29  #
30  # 手番を文字列に
31  #
32  def show_turn():
33      '手番を示す文字列を返す'
34      if turn == FIRST:
35          return('先手')
36      elif turn == SECOND:
37          return('後手')
38      else:
39          return('手番の値が不適切です')
40  #
41  # 手番の初期化
42  #
43  def init_turn():
44      '手番を初期化する'
45      global turn
46      turn = 1
47  #
48  # 手番の交代
49  #
50  def change_turn():
51      '手番を交代する'
52      global turn
53      if turn == FIRST:
54          turn = SECOND
55      elif turn == SECOND:
56          turn = FIRST
57  #
58  # 手番関連の関数のテスト
59  #
60  def test_turn():
61      '手番をテストする'
62      init_turn()
63      print(show_turn(),"の番です")
64      change_turn()
65      print(show_turn(),"の番です")
66      change_turn()
67      print(show_turn(),"の番です")
```

## プログラム 13-3 三目並べのプログラム、実装例（その 3 盤面関連の関数その 1）

```

68  #
69  # 盤面関連の関数
70  #
71  # 盤面を表示する文字列
72  #
73  def show_board():
74      '''盤面を表す文字列を返す'''
75      s = ':012\n-----\n'
76      for i in range(3):
77          s += str(i) + ':'
78          for j in range(3):
79              cell = ''
80              if board[i][j] == OPEN:
81                  cell = ' '
82              elif board[i][j] == FIRST:
83                  cell = '0'
84              elif board[i][j] == SECOND:
85                  cell = 'X'
86              else:
87                  cell = '?'
88              s += cell + '|'
89          s += '\n'
90      return s
91  #
92  # 盤面の初期化
93  #
94  def init_board():
95      '''盤面をすべて空(OPEN)に初期化する'''
96      for i in range(3):
97          for j in range(3):
98              board[i][j] = OPEN
99  #
100 # 盤面の i, j の位置の値を返す
101 #
102 def examine_board(i, j):
103     '''盤面の i 行 j 列の値を返す'''
104     return board[i][j]
105 #
106 # 盤面の i, j に手番 t を登録、状態を文字列で返す
107 #
108 def set_board(i, j, t):
109     '''
110     盤面の i, j に手番 t を登録、状態を文字列で返す
111     返す値は
112     'ok' 成功
113     'Not empty' 空いている場所ではない
114     'illegal turn' 手番が正しくない
115     'illegal slot' 指定された場所が正しくない
116     '''
117     if (i >= 0) and (i < 3) and (j >= 0) and (j < 3):

```

```
118     if (t>0) and (t<3):
119         if examine_board(i,j)==0:
120             board[i][j]=t
121             return 'OK'
122         else:
123             return 'Not empty'
124     else:
125         return 'illegal turn'
126     else:
127         return 'illegal slot'

128 #
129 # 盤面のテスト関数
130 #
131 def test_board():
132     '盤面についてのテストプログラムの1つめです'
133     init_board()
134     print(show_board())
135     print(set_board(0,0,1))
136     print(show_board())
137     print(set_board(1,1,2))
138     print(show_board())
139     print(set_board(1,1,1))
140     print(show_board())
```

## プログラム 13-4 三目並べのプログラム、実装例（その 4 盤面関連の関数その 2）

```
141 #
142 # 水平方向での手番 t の勝ちの判定
143 #
144 def check_board_horizontal(t):
145     '''水平方向に手番 t が勝ちであることを判定します'''
146     for i in range(3):
147         if (board[i][0]==t) and (board[i][1]==t) and (board[i][2]==t):
148             return True
149     return False
150 #
151 # 垂直方向での手番 t の勝ちの判定
152 #
153 def check_board_vertical(t):
154     '''垂直方向に手番 t が勝ちであることを判定します'''
155     for j in range(3):
156         if (board[0][j]==t) and (board[1][j]==t) and (board[2][j]==t):
157             return True
158     return False
159 #
160 # 対角方向での手番 t の勝ちの判定
161 #
162 def check_board_diagonal(t):
163     '''対角方向に手番 t が勝ちであることを判定します'''
164     if (board[0][0]==t) and (board[1][1]==t) and (board[2][2]==t):
165         return True
166     return False
167 #
168 # 逆対角方向での手番 t の勝ちの判定
169 #
170 def check_board_inverse_diagonal(t):
171     '''逆対角方向に手番 t が勝ちであることを判定します'''
172     if (board[0][2]==t) and (board[1][1]==t) and (board[2][0]==t):
173         return True
174     return False
175 #
176 # 手番 t の勝ちの単純な判定
177 #
178 def is_win_simple(t):
179     '''手番 t が勝ちであることを判定します。相手が勝っていることはチェックしません'''
180     if check_board_horizontal(t):
181         return True
182     if check_board_vertical(t):
183         return True
184     if check_board_diagonal(t):
185         return True
186     if check_board_inverse_diagonal(t):
187         return True
188     return False
189 #
```

```
190 # 相手が勝っていないことを確認しての勝ちの判定
191 #
192 def is_win_actual(t):
193     '''手番tが勝ちであることを判定します相手. が勝っていないことも確認します'''
194     if not is_win_simple(t):
195         return False
196     if t==FIRST:
197         if is_win_simple(SECOND):
198             return False
199     else:
200         if is_win_simple(FIRST):
201             return False
202     return True
203 #
204 # 盤面が埋まっていることの判定
205 #
206 def is_full():
207     '''盤面に空きがないことを確認します'''
208     for i in range(3):
209         for j in range(3):
210             if board[i][j]==OPEN:
211                 return False
212     return True
213 #
214 # 引き分けの判定
215 #
216 def is_draw():
217     '''盤面が引き分けであることを判定します'''
218     if is_win_simple(FIRST):
219         return False
220     if is_win_simple(SECOND):
221         return False
222     if not is_full():
223         return False
224     return True
```

## プログラム 13-5 三目並べのプログラム、実装例（その 5 盤面のテスト関数 1）

```
225 #  
226 #盤面のテスト関数 2つめ、勝ち判定のテスト  
227 #  
228 def test_board2():  
229     '''盤面をテストする関数の 2番目'  
230     init_board()  
231     board[0][0]=FIRST  
232     board[1][0]=FIRST  
233     board[2][0]=FIRST  
234     print(show_board())  
235     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))  
236     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))  
237     print("VERTICAL_FIRST:",check_board_vertical(FIRST))  
238     print("VERTICAL_SECOND:",check_board_vertical(SECOND))  
239     init_board()  
240     board[0][0]=SECOND  
241     board[1][0]=SECOND  
242     board[2][0]=SECOND  
243     print(show_board())  
244     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))  
245     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))  
246     print("VERTICAL_FIRST:",check_board_vertical(FIRST))  
247     print("VERTICAL_SECOND:",check_board_vertical(SECOND))  
248  
249     init_board()  
250     board[0][0]=FIRST  
251     board[0][1]=FIRST  
252     board[0][2]=FIRST  
253     print(show_board())  
254     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))  
255     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))  
256     print("VERTICAL_FIRST:",check_board_vertical(FIRST))  
257     print("VERTICAL_SECOND:",check_board_vertical(SECOND))  
258     init_board()  
259     board[0][0]=SECOND  
260     board[0][1]=SECOND  
261     board[0][2]=SECOND  
262     print(show_board())  
263     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))  
264     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))  
265     print("VERTICAL_FIRST:",check_board_vertical(FIRST))  
266     print("VERTICAL_SECOND:",check_board_vertical(SECOND))  
267  
268     init_board()  
269     board[0][0]=FIRST  
270     board[1][1]=FIRST  
271     board[2][2]=FIRST  
272     print(show_board())  
273     print("DIAGONAL_FIRST:",check_board_diagonal(FIRST))  
274     print("DIAGONAL_SECOND:",check_board_diagonal(SECOND))
```

```
275     print("INV_DIAGONAL_FIRST:", check_board_inverse_diagonal(FIRST))
276     print("INV_DIAGONAL_SECOND:", check_board_inverse_diagonal(SECOND))
277     init_board()
278     board[0][0] = SECOND
279     board[1][1] = SECOND
280     board[2][2] = SECOND
281     print(show_board())
282     print("DIAGONAL_FIRST:", check_board_diagonal(FIRST))
283     print("DIAGONAL_SECOND:", check_board_diagonal(SECOND))
284     print("INV_DIAGONAL_FIRST:", check_board_inverse_diagonal(FIRST))
285     print("INV_DIAGONAL_SECOND:", check_board_inverse_diagonal(SECOND))
286
287     init_board()
288     board[0][2] = FIRST
289     board[1][1] = FIRST
290     board[2][0] = FIRST
291     print(show_board())
292     print("DIAGONAL_FIRST:", check_board_diagonal(FIRST))
293     print("DIAGONAL_SECOND:", check_board_diagonal(SECOND))
294     print("INV_DIAGONAL_FIRST:", check_board_inverse_diagonal(FIRST))
295     print("INV_DIAGONAL_SECOND:", check_board_inverse_diagonal(SECOND))
296     init_board()
297     board[0][2] = SECOND
298     board[1][1] = SECOND
299     board[2][0] = SECOND
300     print(show_board())
301     print("DIAGONAL_FIRST:", check_board_diagonal(FIRST))
302     print("DIAGONAL_SECOND:", check_board_diagonal(SECOND))
303     print("INV_DIAGONAL_FIRST:", check_board_inverse_diagonal(FIRST))
304     print("INV_DIAGONAL_SECOND:", check_board_inverse_diagonal(SECOND))
```

## プログラム 13-6 三目並べのプログラム、実装例（その 6 盤面のテスト関数 2）

```

305 #
306 #盤面のテスト関数 3、勝ち、引き分けの判定
307 #
308 def test_board3():
309     '''盤面をテストする関数の3番目'''
310     init_board()
311     board[0][0]=FIRST
312     board[1][0]=FIRST
313     board[2][0]=SECOND
314     board[0][1]=SECOND
315     board[1][1]=SECOND
316     board[2][1]=FIRST
317     board[0][2]=FIRST
318     board[1][2]=FIRST
319     board[2][2]=SECOND
320     print(show_board())
321     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))
322     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))
323     print("VERTICAL_FIRST:",check_board_vertical(FIRST))
324     print("VERTICAL_SECOND:",check_board_vertical(SECOND))
325     print("DIAGONAL_FIRST:",check_board_diagonal(FIRST))
326     print("DIAGONAL_SECOND:",check_board_diagonal(SECOND))
327     print("INV_DIAGONAL_FIRST:",check_board_inverse_diagonal(FIRST))
328     print("INV_DIAGONAL_SECOND:",check_board_inverse_diagonal(SECOND))
329     print("IS_WIN_SIMPLE_FIRST",is_win_simple(FIRST))
330     print("IS_WIN_SIMPLE_SECOND",is_win_simple(SECOND))
331     print("IS_WIN_ACTUAL_FIRST",is_win_actual(FIRST))
332     print("IS_WIN_ACTUAL_SECOND",is_win_actual(SECOND))
333     print("IS_FULL",is_full())
334     print("IS_DRAW",is_draw())
335
336     init_board()
337     board[0][0]=FIRST
338     board[1][0]=SECOND
339     board[2][0]=FIRST
340     board[0][1]=SECOND
341     board[1][1]=FIRST
342     board[2][1]=OPEN
343     board[0][2]=FIRST
344     board[1][2]=OPEN
345     board[2][2]=SECOND
346     print(show_board())
347     print("HORIZONTAL_FIRST:",check_board_horizontal(FIRST))
348     print("HORIZONTAL_SECOND:",check_board_horizontal(SECOND))
349     print("VERTICAL_FIRST:",check_board_vertical(FIRST))
350     print("VERTICAL_SECOND:",check_board_vertical(SECOND))
351     print("DIAGONAL_FIRST:",check_board_diagonal(FIRST))
352     print("DIAGONAL_SECOND:",check_board_diagonal(SECOND))
353     print("INV_DIAGONAL_FIRST:",check_board_inverse_diagonal(FIRST))
354     print("INV_DIAGONAL_SECOND:",check_board_inverse_diagonal(SECOND))

```

```
355     print("IS_WIN_SIMPLE_FIRST", is_win_simple(FIRST))
356     print("IS_WIN_SIMPLE_SECOND", is_win_simple(SECOND))
357     print("IS_WIN_ACTUAL_FIRST", is_win_actual(FIRST))
358     print("IS_WIN_ACTUAL_SECOND", is_win_actual(SECOND))
359     print("IS_FULL", is_full())
360     print("IS_DRAW", is_draw())
361
362     init_board()
363     board[0][0] = SECOND
364     board[1][0] = FIRST
365     board[2][0] = SECOND
366     board[0][1] = FIRST
367     board[1][1] = SECOND
368     board[2][1] = FIRST
369     board[0][2] = SECOND
370     board[1][2] = OPEN
371     board[2][2] = FIRST
372     print(show_board())
373     print("HORIZONTAL_FIRST:", check_board_horizontal(FIRST))
374     print("HORIZONTAL_SECOND:", check_board_horizontal(SECOND))
375     print("VERTICAL_FIRST:", check_board_vertical(FIRST))
376     print("VERTICAL_SECOND:", check_board_vertical(SECOND))
377     print("DIAGONAL_FIRST:", check_board_diagonal(FIRST))
378     print("DIAGONAL_SECOND:", check_board_diagonal(SECOND))
379     print("INV_DIAGONAL_FIRST:", check_board_inverse_diagonal(FIRST))
380     print("INV_DIAGONAL_SECOND:", check_board_inverse_diagonal(SECOND))
381     print("IS_WIN_SIMPLE_FIRST", is_win_simple(FIRST))
382     print("IS_WIN_SIMPLE_SECOND", is_win_simple(SECOND))
383     print("IS_WIN_ACTUAL_FIRST", is_win_actual(FIRST))
384     print("IS_WIN_ACTUAL_SECOND", is_win_actual(SECOND))
385     print("IS_FULL", is_full())
386     print("IS_DRAW", is_draw())
```

## プログラム 13-7 三目並べのプログラム、実装例（その 7 棋譜関連の関数）

```

387  #
388  # ログのリプレイ
389  #
390  def replay_log(log):
391      ''' 棋譜 log をたどります。print 文で画面に出力します '''
392      init_board()
393      init_turn()
394      print(show_board())
395      for m in log:
396          if len(m) == 2:
397              print(show_turn(), "の番です")
398              print(set_board(m[0], m[1], turn))
399              print(show_board())
400              print("IS_WIN:", turn, ": ", is_win_actual(turn))
401              change_turn()
402          else:
403              print("RESULT_IN_LOG:", m[0])
404              print("IS_WIN_FIRST:", is_win_actual(FIRST))
405              print("IS_WIN_SECOND:", is_win_actual(SECOND))
406              print("IS_DRAW:", is_draw())
407  #
408  # ログのテスト
409  #
410  def test_log():
411      ''' 棋譜をテストします '''
412      print("LOG1")
413      replay_log(log1)
414      print("LOG2")
415      replay_log(log2)
416      print("LOG3")
417      replay_log(log3)
418  #
419  # すべてのテスト
420  #
421  def test_all():
422      ''' すべてのテストを行います '''
423      test_turn()
424      test_board1()
425      test_board2()
426      test_board3()
427      test_log()

```

### プログラム 13-8 三目並べのプログラム、実装例（その 8 プレイ関数とメインプログラム）

```

428 #
429 # 実際のプレイ
430 #
431 def play():
432     '''端末への入出力を用いて実際に三目並べをプレイする関数です'''
433     init_turn()
434     init_board()
435     print(show_board())
436     # 棋譜用の空リストを作る。play() の外側でアクセスするなら global 売言
437     # global log
438     log = []
439     while True:
440         print(show_turn(), "の番です")
441         while True:
442             row = int(input("行を入力してください:"))
443             column = int(input("列を入力してください:"))
444             result = set_board(row, column, turn)
445             print(result)
446             if result == "OK":
447                 break
448             print("不適切な入力です。再度、入力して下さい")
449             # ここ(内側の while の外)で log に手を追加
450             #
451             # 要追加
452             #
453             print(show_board())
454             if is_draw():
455                 print("引き分けです")
456                 # ここで棋譜に勝敗_(引き分け)_を追加
457                 break
458             if is_win(actual(turn)):
459                 print(show_turn(), "の勝ちです")
460                 # ここで棋譜に勝敗_(turnの勝ち)_を追加
461                 break
462             change_turn()
463             # ここで棋譜のリプレイ
464             # 現在は log は空なので判定して処理
465             if len(log) > 0:
466                 replay_log(log)
467             else:
468                 print("棋譜は作成されていません")
469                 if __name__ == '__main__':
470                     print('三目並べ')

```

### 3) プログラムのテスト

上記のプログラムを読み、Python Shell で実行した後、シェルからテスト用の関

数を実行してプログラムの動作を確認してください。用意されているテスト関数は上から順に以下のものです。

```
test_turn()  
test_board1()  
test_board2()  
test_board3()  
test_log()  
test_all()
```

#### 4) プログラムの実行

上記のプログラムを実行後、Python Shell から play() 関数を呼び出して実際に三目並べを行ってみてください。

```
play()
```

#### 演習 13-2 棋譜の採取

上記のプログラムを拡張し、play() 関数の中で、そのプレイの棋譜を採取するようにしてください。また勝敗が確定した後、棋譜を再生するようにしてください。

## 13.6 力試し

これまでに学んだ方法を使い、力試しとして、以下のようなことに挑戦してください。

- CUI ではなく、tkinter を用いて GUI で動く三目並べを作る
- 手番や盤面、棋譜などをそれを操作する関数をメソッドとしてクラスで実装する。
- 人間同士ではなく、片側をコンピュータがプレイするように拡張する。
- 三目並べではなく、オセロゲームについて同様のプログラムを開発する。

## 13.7 プログラムの開発に関連するいくつかの話題

### 13.7.1 テストで分かること

コンピュータのプログラムは通常、無限ともいえる場合に適正に稼働しなければなりません。例えば三目並べでも可能な棋譜の数は相当に多いものです。このため、テストについては

- テストにパスしなかったプログラムは誤りがあることは容易に分かりますが
  - テストにパスしたことが期待されるすべての場合にプログラムが適切に稼働することは保証しない
- ということにも留意しましょう。

部品となる関数のテストは、それらを組み合わせた関数のテストよりも行いやすく、正しく動作する部品で作ることで複雑な関数への信頼度が高まります。

### 13.7.2 リファクタリング

プログラムの改善には以下の2つの方向があります。

- プログラムの機能を強化する
  - プログラムの機能は維持し、実装を整理して維持管理や拡張を容易にすること
- 後者のような改善を「リファクタリング」と呼びます。例えば、三目並べのプログラムについてクラスを用いて実装しなおす、などはこれにあたります。リファクタリングが必要とされる理由はいくつか挙げられます。
- プログラムは長期にわたって使われるため、保守が容易でなければならない。
  - プログラムの開発者は入れ替わる可能性がある。
  - プログラムへの機能追加などの要求が継続的に発生する。

### 13.7.3 ソフトウェア開発のVモデル

上記の三目並べのような小さな例でも、プログラムの開発の前半は、何を作るべきか（要求仕様）を明確にし、それを段階的に詳細化して実装する関数などを明確にしています。後半は、相互依存のない関数から順に実装とテストを繰り返し、全体を積み上げて完成に至ります。

このようなプロセスは下の図のようなV型で表され、ソフトウェア開発のV

モデルと呼ばれます。

V型の構造のため、全体に近いV型の上方では、設計から実装・テストまではその下に多くの作業が含まれ、距離が遠くなります。このため上方での設計の不適切さは実装とテストで発見されるまで時間がかかり、手戻りによる手直しも多くなり慎重に行う必要があります。

ソフトウェア開発を成功させる方向として以下の2つの方向が考えられます。後者はアジャイル開発などと呼ばれる考え方です。

- 要求仕様をしっかりと定め、手戻りを減らす
- 要求仕様を絞り込みさっさと開発して、必要なら追加する

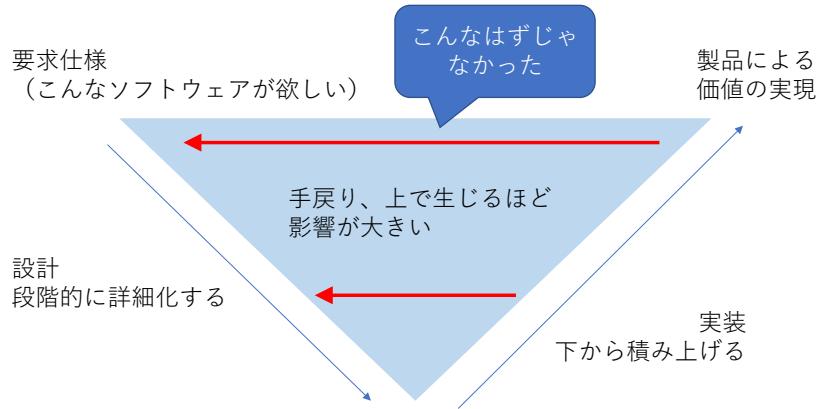


図 13-5 ソフトウェア開発の V モデル

# 14. Python の学術利用

## 14.1 本章の学習の目標

Python が注目されている理由の一つは数値計算など学術利用に適したライブラリが豊富に提供されていることです。本章では以下の 3 つのライブラリについて、NumPy や pandas でのデータの扱い方と、Matplotlib でのグラフ描画の基礎を学びます。

それぞれ、かなり高機能なパッケージですし、適用領域の知識も必要になりますので、ここでは基礎的な事項だけ学びます。

1. NumPy: 科学技術領域での数値計算のための基礎的なパッケージです  
SciPy: より高度な数値計算のためのライブラリです。
2. Matplotlib: データをグラフにプロットするためのパッケージです
3. pandas: データ分析のためのパッケージです

これらは次の図のように相互に関連しています。

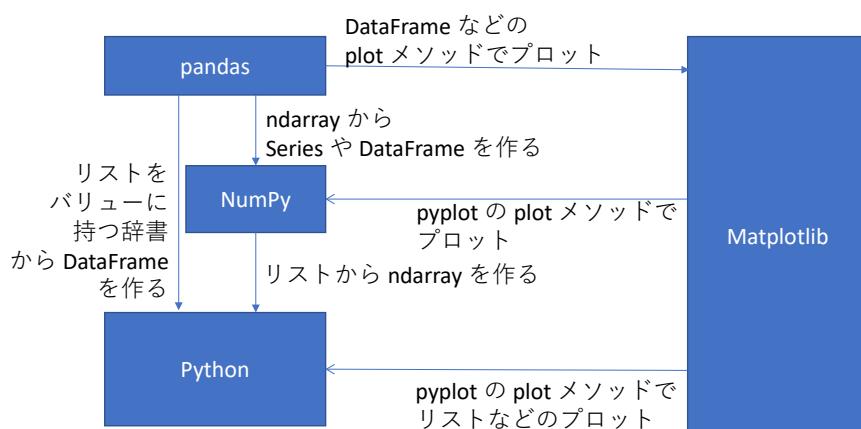


図 14-1 NumPy, Matplotlib, pandas の関連

## 14.2 import 時の別名

NumPy, Pandas, Matplotlib などの import には以下のような別名がよく使われます。本書でもこれを用います。

```

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

```
import pandas as pd
```

## 14.3 NumPy

Python はそれ自体は実行の遅いプログラミング言語ですが、NumPy の内部では C 言語で実装されており、ベクトルや行列などの演算が高速で実行できます。

### 14.3.1 多次元配列の生成

Python でデータを一括して扱うにはリストを使いますが、NumPy では固有のデータ形式である ndarray（別名 array）を使います。

#### 1) リストから生成する

```
import numpy as np
data1 = [1, 2, 3]
arr1 = np.array(data1) # 1 次元のデータ
data2 = [[1,2,3],[4,5,6]]
arr2 = np.array(data2)
```

#### 2) すべての要素が 0 の配列を作る

```
np.zeros(5) # 大きさ 5 の 1 次元配列
array([0., 0., 0., 0., 0.])
np.zeros((2,2)) # 大きさ (2,2) の 2 次元配列, () が二重に注意
array([[0., 0.],
       [0., 0.]])
a = np.array([[1,2,3],[4,5,6]])
np.zeros_like(a) # 配列 a と同じ大きさ
array([[0, 0, 0],
       [0, 0, 0]])
```

すべての要素が 1 の配列は同様に ones, ones\_like で作れます。

### 14.3.2 ndarray の属性

ndarray では以下のようないくつかの属性をしらべることができます。

- ndim : ndarray の次元

- shape: ndarray の大きさ
- dtype: データ型

```
import numpy as np
arr2 = np.array([[1,2,3],[4,5,6]]
arr2.ndim
2
arr2.shape
(2,3)
arr2.dtype
dtype('Int32')
```

なお、Python の整数型は桁数の制限がないものでしたが、ndarray では計算を高速に実施するため高速演算が可能な固定長の型が採用されています。

### 14.3.3 ndarray の要素へのアクセス

ndarray の要素はリストと同じように [] でアクセスできます。添え字の 0 始まりも同じです。

```
arr1 = np.array([1,2,3])
arr1[0]
1
arr1[1] = 1
arr1
array([1,1,3])
```

多次元配列では [][] の代わりに [,] という記法も使えます。

```
arr2[0][0]
arr2[0,0]
```

### 14.3.4 スライス

ndarray でもリスト同様、スライスが使えます。

```
arr1[2:]
array([3])
```

多次元配列では[:,:] という記法を用います。

```
arr2[0:2,0:2]
array([[1,2],[4,5]])
```

**注意** ndarray のスライスの結果は「コピー」ではなく、もとの ndarray の一部への参照です。

スライスにスカラー値を代入するとすべての要素に代入されます。

### 14.3.5 ndarray の演算

ndarray 型のデータに対しては四則演算、べき乗、比較などができます。これらは要素ごとの演算になります。

行列の積は @ という演算を使います。

スカラー値との演算では、すべての要素にその値が適用されます。

```
arr1 = np.array([1,2,3])
arr1*2
array([2,4,6])
arr1 + 1
array([2,3,4])
```

### 14.3.6 条件を満たす要素の抽出

以下の方法で条件を満たす要素を抽出できます。

```
arr1 = np.array([1,2,3,4,5])
cond = arr1 > 2 # 条件を満たすかどうかの配列の生成
cond
array([False, False,  True,  True,  True])
arr1[cond]          # 条件を指定してスライス
array([3, 4, 5])
```

### 14.3.7 行列計算

Numpy では ndarray を対象に数学の行列計算を簡単に実行できます。

- 行列の転置（行と列の入れ替え）  
 ndarray の T 属性を使います。
- 行列積  
 @ 演算を使います。
- linalg (linear algebra, 線形代数) モジュールの利用：numpy.linalg (numpy を np という別名にしてインポートしていれば np.linalg) には以下のような行列を扱う関数が定義されています。

- `diag`(対角要素),
- `trace`(対角要素の和) ,
- `det`(行列式),
- `eig`(固有値),
- `inv`(逆行列),
- `solve`(一次方程式を解く)

### 14.3.8 亂数

Numpy では一括して乱数を生成することができます.

- `seed` 亂数生成の初期値を設定します
- `rand` 一様連続乱数を生成します
- `randn` 標準正規分布に従う乱数を生成します
- `randint` 与えられた範囲の乱数を生成します

以下が使用例です

- `np.random.rand(10)`  
0 から 1 までの浮動小数点乱数を 10 個発生
- `np.random.randn(5,5)`  
大きさ (5,5) の 2 次元配列として標準正規分布に従う乱数を発生
- `np.random.permutation([1,2,3,4,5])`  
リスト [1,2,3,4,5] のランダムな並べ替えを生成. 引数には `range()` や `ndarray` も指定可. 多次元配列では最初の添え字のみ入れ替え
- `np.random.randint(2, size=10)`  
(0 以上) 2 未満の整数乱数を大きさ 10 で生成. 下限, 上限を与えることが可能なため, 大きさは `size =` として指定

## 14.4 Matplotlib

### 14.4.1 backend: グラフ出力環境

Matplotlib ではグラフの出力方法を選ぶことが可能です. ここでは IDLE で動かすため `tkagg` という `tkinter` を利用した環境の指定方法を説明します.

- Matplotlib ではグラフを実際に出力する環境のことを `backend` と呼びます. さまざまな `backend` が用意されています.
- `tkagg` は Tkinter を使ってグラフを出力する `backend` です.

- IDLE 環境では、matplotlib のインポートのあとに `use()` 関数を使って指定します。

```
import matplotlib
matplotlib.use('tkagg')
```

- `use()` の指定はプロット用のモジュール `matplotlib.pyplot` のインポートより前にしなければなりません。

- Ipython やそれを使う Jupyter Notebook では `matplotlib` の利用に先だって使用したい `backend` に応じて以下を指定します。

```
%matplotlib notebook
%matplotlib tk
```

#### 14.4.2 日本語での文字出力

- `matplotlib` の標準のフォントは日本語文字を持たないので表示は□になってしまいます。
- `matplotlib` version 3.1 以降では ttc フォントが使えるようになりました。これ以降のバージョンをご利用の方はフォントの追加インストールは不要です<sup>1</sup>。
- 指定方法はいくつかありますが、ここではプログラム内で一括指定する方法を示します。
- フォントを追加でインストールした場合はユーザのフォルダにある `.matplotlib` フォルダの `fontList.json` が古ければいったん消去します。
- 利用法の概要

```
import matplotlib
# Set tkinter as the output destination before importing pyplot
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
# Enable matplotlib to display Japanese characters.
# Yu Gothic can be used in matplotlib version 3.1 or later
matplotlib.rc('font', **{'family':'Yu Gothic'})
# For Mac User, try the following instead of the above line
# matplotlib.rc('font', **{'family':'Hiragino Maru Gothic Pro'})
# The following is an example plot
data = [1,2,3]
```

---

<sup>1</sup> 以下の例では `Yu Gothic` を設定していますが `Yu Mincho`, `MS Gothic`, `MS Mincho` なども使えるはずです。

```
plt.plot(data)
plt.title('Title')
plt.show()
```

### 14.4.3 タイトル, 軸ラベル, 線種の設定

- グラフのタイトルを設定する関数は title です.
- X-軸ラベルを設定する関数は xlabel です.
- Y-軸ラベルを設定する関数は ylabel です.
- 線種の指定は plot 関数の引数で与えます.
  - 色, 線種, マーカの文字列で指定する方法
  - color, linestyle, linewidth などの引数で指定する方法.
- 使用例（該当部分）

```
plt.plot([1,2,3], 'k-') #黒の実線
plt.plot([2,3,4], 'r--') #赤の破線
plt.plot([3,4,5], 'b--o') #青の破線, ○のマーカ
plt.title('タイトル')
plt.xlabel('横軸')
plt.ylabel('縦軸')
plt.show()
```

### 14.4.4 利用例

#### 1) use\_matplotlib.py

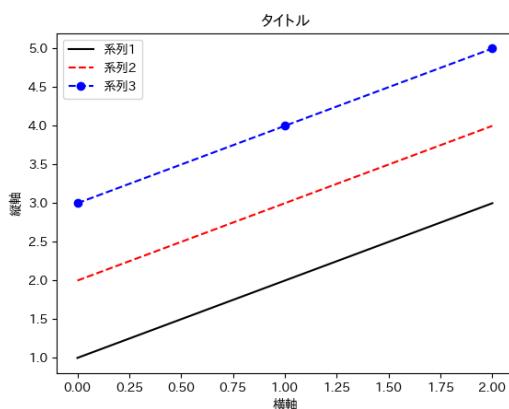


図 14-2 Matplotlib の使用例

## プログラム 14-1 use\_matplotlib.py

行	ソースコード
1	#
2	#matplotlib 基本の使い方
3	#
4	import matplotlib
5	#
6	#出力先としてtkinterをuseで設定、pyplotのインポートより前に
7	#
8	matplotlib.use('tkagg')
9	import matplotlib.pyplot as plt
10	#
11	#matplotlibで日本語表示を可能にする
12	#使用環境で適当なもののコメントを外す
13	#
14	#For Windows
15	matplotlib.rcParams['font', **{'family': 'Yu Gothic'})
16	#For Campus PC Terminal
17	#matplotlib.rcParams['font', **{'family': 'IPAPGothic'})
18	#For macOS
19	#matplotlib.rcParams['font', **{'family': 'Hiragino Maru Gothic Pro'})
20	#
21	#3本の線グラフを書く
22	#
23	plt.plot([1,2,3], 'k-', label='系列 1')
24	plt.plot([2,3,4], 'r--', label='系列 2')
25	plt.plot([3,4,5], 'b--o', label='系列 3')
26	#
27	plt.title('タイトル')
28	plt.xlabel('横軸')
29	plt.ylabel('縦軸')
30	plt.legend() #凡例
31	plt.show()

## 2) 散布図の描画

散布図は pyplot の scatter 関数に x 軸データ, y 軸データを与えて描画します.

- use\_matplotlib\_scatter.py

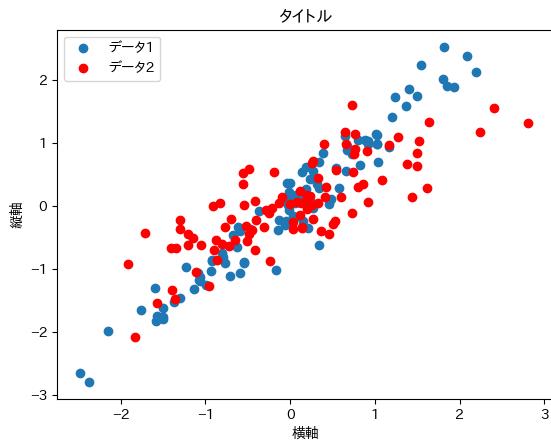


図 14-3 散布図の描画

## プログラム 14-2 use\_matplotlib\_scatter.py

行	ソースコード
1	#
2	#_matplotlib_で散布図を描く
3	#
4	import matplotlib
5	#
6	matplotlib.use('tkagg')
7	import matplotlib.pyplot as plt
8	import numpy as np
9	#
10	#_matplotlib_で日本語表示を可能にする
11	#_使用環境で適当なもののコメントを外す
12	#
13	#_For_Windows
14	matplotlib.rcParams['font', **{'family': 'YuGothic'}]
15	#_For_Campus_PC_Terminal
16	#matplotlib.rcParams['font', **{'family': 'IPAPGothic'}]
17	#_For_macOS
18	#matplotlib.rcParams['font', **{'family': 'Hiragino_Maru_Gothic_Pro'}]
19	#
20	#_ランダムなデータの作成
21	#
22	datax=np.random.randn(100)
23	datay=datax+np.random.randn(100)*0.3
24	#
25	#_散布図の描画
26	#
27	plt.scatter(datax,datay,label='データ 1')
28	#
29	#_別のデータの作成
30	#
31	datax=np.random.randn(100)
32	datay=0.6*datax+np.random.randn(100)*0.4
33	#

```

34 #_色を指定して散布図を作成
35 #
36 plt.scatter(datax,datay,color='red',label='データ 2')
37 #
38 #_タイトル,_軸ラベル,_凡例の記入
39 #
40 plt.title('タイトル')
41 plt.xlabel('横軸')
42 plt.ylabel('縦軸')
43 plt.legend()
44 #
45 #_表示
46 #
47 plt.show()

```

•

### 3) ヒストグラムの描画

ヒストグラムは pyplot の hist 関数にデータを与えて描画します。階級数などは自動調整されますが、指定することも可能です。

use\_matplotlib\_hist.py

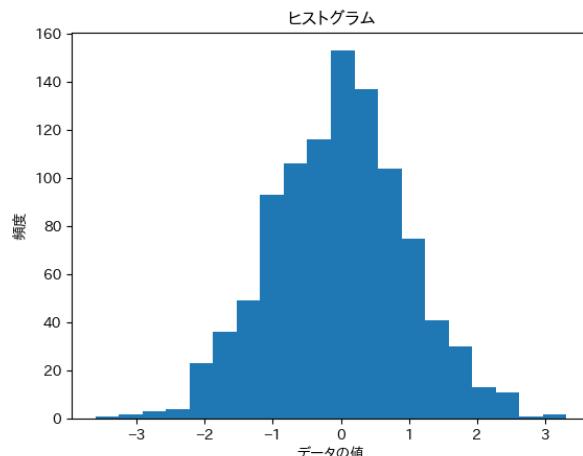


図 14-4 ヒストグラムの描画

プログラム 14-3 use\_matplotlib\_hist.py

行	ソースコード
1	#
2	#_matplotlib_でヒストグラムを描く
3	#
4	import matplotlib
5	#
6	#_出力先としてtkinterを設定,_pyplot_のインポートより前に
7	#
8	matplotlib.use('tkagg')
9	import matplotlib.pyplot as plt

```
10 import numpy as np
11 #
12 # matplotlib で日本語表示を可能にする
13 # 使用環境で適当なもののコメントを外す
14 #
15 # For Windows
16 matplotlib.rcParams['font', **{'family': 'Yu Gothic'})
17 # For Campus PC Terminal
18 #matplotlib.rcParams['font', **{'family': 'IPAPGothic'}]
19 # For macOS
20 #matplotlib.rcParams['font', **{'family': 'Hiragino Maru Gothic Pro'}]
21 #
22 # ヒストグラムの作成
23 #
24 data = np.random.randn(1000)
25 plt.hist(data, bins=20)
26 #
27 # タイトル、軸ラベルを設定
28 #
29 plt.title('ヒストグラム')
30 plt.xlabel('データの値')
31 plt.ylabel('頻度')
32 #
33 # 表示
34 #
35 plt.show()
```

#### 4) 複数グラフの描画

Matplotlib では以下の方法で複数のグラフを並べて描画できます。

- pyplot の figure 関数で Figure オブジェクトを得ます。
- Figure オブジェクトに pyplot の add\_subplot 関数で subplot を追加します。  
結果を変数に保存します。
- subplot の間隔を pyplot の subplots\_adjust 関数で調整します。
- 各 subplot に plot, scatter, hist 関数で描画します。
- タイトル、軸ラベルは set\_title, set\_xlabel, set\_ylabel で追加。関数名に注意。

use \_matplotlib \_subplot.py

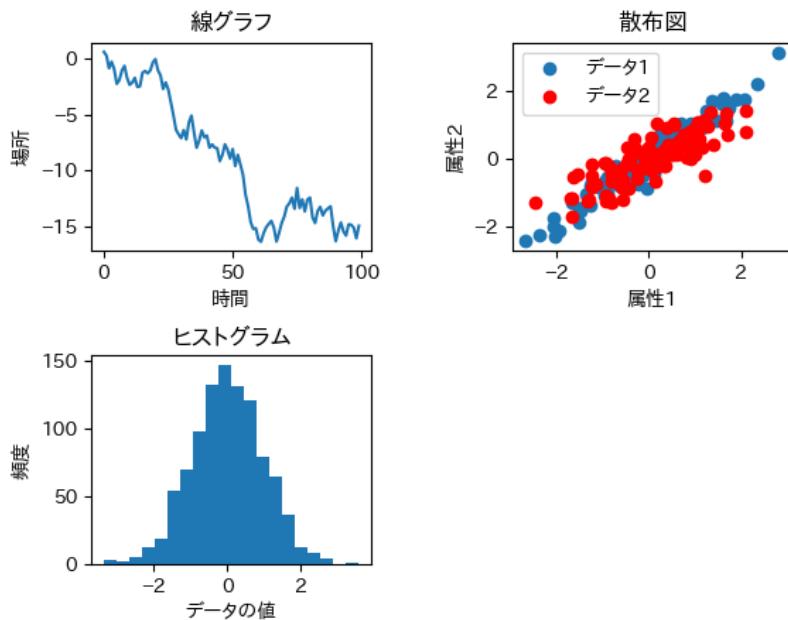


図 14-5 複数のグラフの描画

## プログラム 14-4 use\_subplot.py

行	ソースコード
1	#
2	#_subplot_を使う例
3	#
4	import matplotlib
5	matplotlib.use('tkagg')
6	import matplotlib.pyplot as plt
7	import numpy as np
8	#
9	#_テキストで日本語を表示できるようにする
10	#_使用環境で適当なもののコメントを外す
11	#_For_Windows
12	matplotlib.rcParams['font', **{'family': 'Yu Gothic'})
13	#_For_Campus_PC_Terminal
14	#matplotlib.rcParams['font', **{'family': 'IPAPGothic'})
15	#_For_macOS
16	#matplotlib.rcParams['font', **{'family': 'Hiragino_Maru_Gothic_Pro'})
17	#
18	#
19	#_3_つの subplot_を作成,_間隔を調整
20	#
21	fig=plt.figure()
22	ax1=fig.add_subplot(2,2,1)
23	ax2=fig.add_subplot(2,2,2)
24	ax3=fig.add_subplot(2,2,3)
25	plt.subplots_adjust(hspace=0.5, wspace=0.5)
26	#
27	#_1_つめに線グラフを出力

```
28 #_
29 data_=np.random.randn(100).cumsum()
30 ax1.plot(data)
31 ax1.set_title('線グラフ')
32 ax1.set_xlabel('時間')
33 ax1.set_ylabel('場所')
34 #
35 #_2つめに散布図を出力
36 #
37 datax_=np.random.randn(100)
38 datay_=datax_+np.random.randn(100)*0.3
39 ax2.scatter(datax,datay,label='データ 1')
40
41 datax_=np.random.randn(100)
42 datay_=0.6*datax_+np.random.randn(100)*0.4
43 ax2.scatter(datax,datay,color='red',label='データ 2')
44
45 ax2.set_title('散布図')
46 ax2.set_xlabel('属性 1')
47 ax2.set_ylabel('属性 2')
48 ax2.legend()
49
50 #
51 #_3つめにヒストグラムを出力
52 #
53 data_=np.random.randn(1000)
54 ax3.hist(data,bins=20)
55
56 ax3.set_title('ヒストグラム')
57 ax3.set_xlabel('データの値')
58 ax3.set_ylabel('頻度')
59
60 #
61 #_グラフを表示
62 #
63 plt.show()
```

## 14.5 pandas

### 14.5.1 Dataframe

Pandas 固有のデータ形式には以下のものがあります。

- 1 次元の Series
- 2 次元の DataFrame

DataFrame には行名(index)と列名(column)がつけられます。



## 14.5.2 DataFrame を作る

### 1) numpy の array から作る

```
import numpy as np
import pandas as pd
d = np.array([[1,2,3],[4,5,6],[7,8,9]])
df = pd.DataFrame(d,columns=['a','b','c'])
df
   a   b   c
0  1   2   3
1  4   5   6
2  7   8   9
```

列名と行名はそれぞれ

```
df.columns
```

```
df.index
```

で調べられます。

### 2) リスト型を値にもった辞書から作る

```
df = pd.DataFrame({'a': [1,4,7], 'b':[2,5,8], 'c':[3,6,9]})
df
   a   b   c
0  1   2   3
```

1 4 5 6

2 7 8 9

辞書は「キー」と「値（バリュー）」のペアの集合で作られる Python のデータ型です。

```
dic = {'a':1, 'b':2, 'c':3}
```

キーから値を検索できます。

```
dic['a']
```

1

### 14.5.3 csv ファイルを読み込む

- csv ファイル形式で保存された表計算ソフトのデータを読み込んで DataFrame を作れます

```
df = pd.read_csv(ファイル名)
```

- 先頭行が列名として扱われます。

➤ すべてをデータとして読むときには、オプションとして以下を指定します。

```
header = None か names = [列名のリスト]
```

- Windows で日本語を含むファイルを読み込む場合は漢字コードを指定します。

```
encoding = 'SHIFT-JIS'
```

- 列名に漢字コードを使うと、列名でのデータの指定などでエラーが出ます。

- sample.csv は以下のようなファイルです。

ID	Japanese	English	Mathematics	Science	Social Studies
A	91	69	100	82	94
B	80	60	45	52	46
C	92	92	76	73	97
D	58	50	60	71	77
E	58	75	96	96	94
F	92	89	86	82	74
G	97	87	59	55	56

以下の手順で読み込みます (use\_read\_csv.py)

- numpy のインポート
- pandas のインポート
- os のインポート

4. フォルダのパスを得る

```
folderpath = input("Enter folder path:")
```

5. os.chdir で csv ファイルのあるフォルダに移動

```
os.chdir(フォルダのファイルパス)
```

6. CSV ファイルの読み込み

```
df = pd.read_csv("ファイル名")
```

注意：pd.read\_csv は日本語のファイル名を正しく処理できないようです。

**プログラム 14-5 use\_read\_csv.py**

行	ソースコード
1	import numpy as np
2	import pandas as pd
3	import os
4	#
5	# データあるフォルダに移動, 文字列に r を付けて特殊な文字の解釈を止める
6	#
7	# pandas は日本語のファイル名をうまく処理できない
8	#
9	folderpath = input("Enter folder path:")
10	os.chdir(folderpath)
11	df = pd.read_csv("sample.csv")
12	#
13	# Sum horizontally (axis=1) and create a column called Total
14	df['Total'] = df.sum(axis=1)
15	# Display the DataFrame df
16	print(df)
17	# Display summary statistics for DataFrame df
18	print(df.describe())

#### 14.5.4 要約統計量の表示

describe メソッドで要約統計量を表示できます。

#### 14.5.5 Pandas のデータのプロット

Pandas でのプロットは DataFrame 側の plot メソッドを呼び出します。

(use\_DataFrame\_plot.py)

```
df.plot()      # 折れ線グラフ
df.plot.bar(stacked=True) # 積み上げ棒グラフ
df.plot.scatter('Japanese','English') # 列を指定して散布図
df['Japanese'].plot.hist() # 列を指定してヒストグラム
```

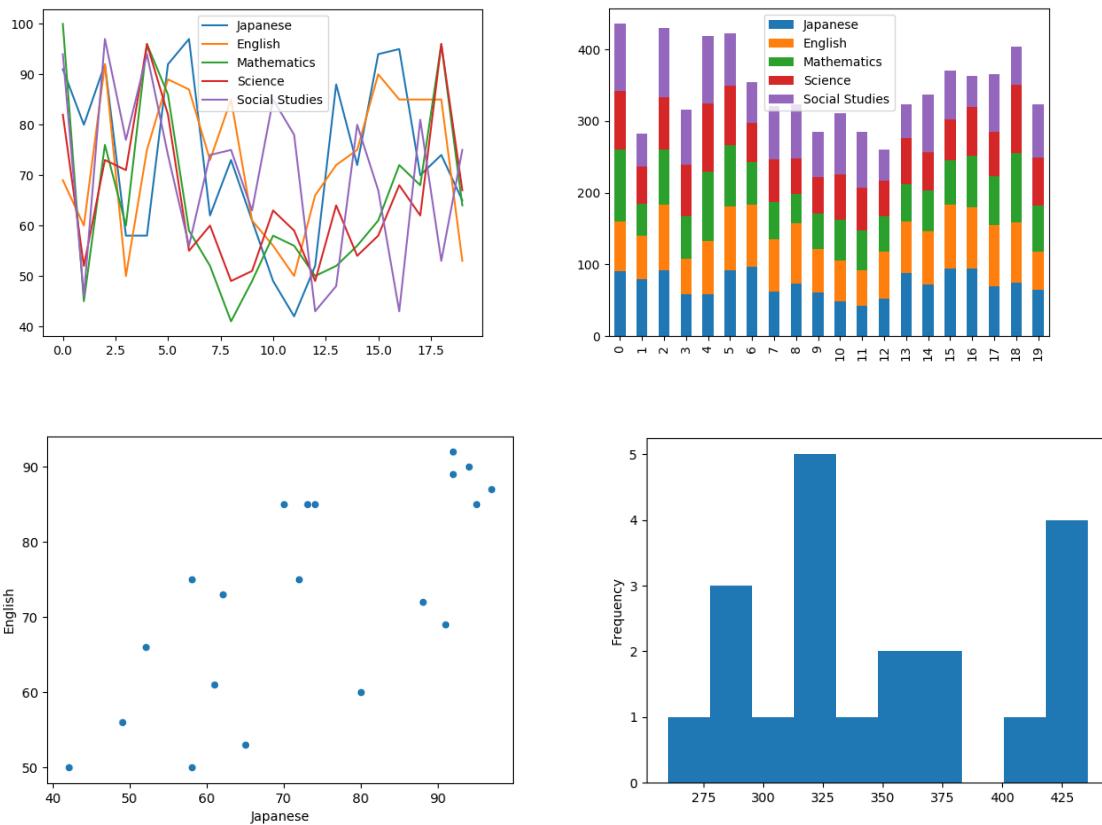


図 14-6 pandas でのグラフ作成

## プログラム 14-6 use\_DataFrame\_plot.py

行	ソースコード
1	import numpy as np
2	import pandas as pd
3	import os
4	import matplotlib
5	matplotlib.use('tkagg')
6	import matplotlib.pyplot as plt
7	#
8	#_データあるフォルダに移動
9	#
10	#_フォルダのパスを得る
11	folderpath_=input("Enter_folder_path:")
12	os.chdir(folderpath)
13	#
14	df_=pd.read_csv("sample.csv")
15	#
16	#
17	#_折れ線グラフ
18	#
19	

```
20 df.plot()
21 print("次に進むにはグラフィンドウを閉じてください")
22 plt.show()
23 #
24 # 積み上げ棒グラフ
25 #
26 df.plot.bar(stacked=True)
27 print("次に進むにはグラフィンドウを閉じてください")
28 plt.show()
29 #
30 # 散布図
31 #
32 df.plot.scatter('Japanese','English')
33 print("次に進むにはグラフィンドウを閉じてください")
34 plt.show()
35 #
36 # 水平方向(axis=1)に総和をとり '_Total' という列を作る
37 #
38 df['Total']=df.sum(axis=1)
39 #
40 #
41 # ヒストグラム
42 #
43 df['Total'].plot.hist()
44 print("次に進むにはグラフィンドウを閉じてください")
45 plt.show()
```

## 14.6 課題

np\_matplotlib.py は Numpy と matplotlib を使って 1 乗~4 乗のグラフを描くプログラムです。

### 演習 14-1 Matplotlib でのこぎり波のフーリエ近似の描画

プログラム 14-7 を改造してのこぎり波のフーリエ近似を描くプログラムを作成しなさい。

Numpy (np) では円周率は np.pi, 正弦関数は np.sin() で利用できます。

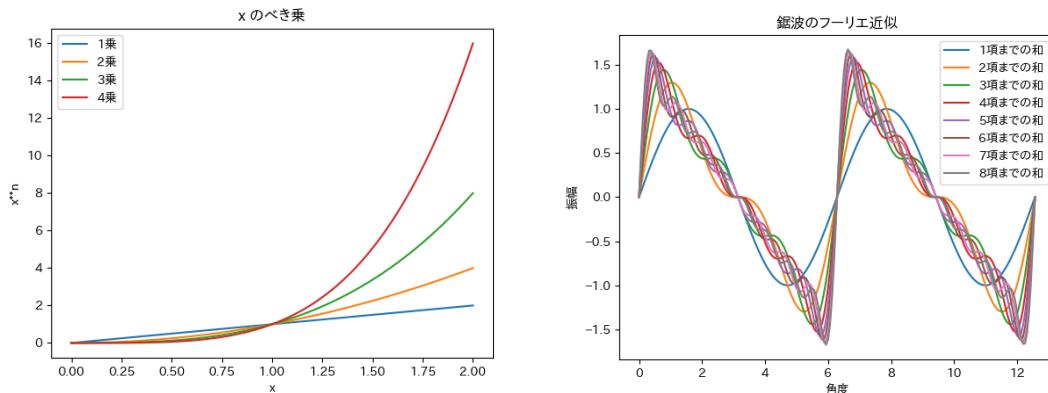


図 14-7 べき乗のグラフとのこぎり波の三角関数の和での近似

プログラム 14-7 Numpy と Matplotlib で 1~4 乗のグラフを描く

(use\_matplotlib\_power\_function.py)

行	ソースコード	説明
1	# Numpy のデータを plot する例題	
2	import matplotlib	matplotlib の準備
3	matplotlib.use('tkagg')	
4	import matplotlib.pyplot as plt	numpy の準備
5	import numpy as np	matplotlib のフォント指定
6	# For Windows	
7	matplotlib.rc('font', **{'family': 'Yu_Gothic'})	
8	# For Campus PC Terminal	
9	#matplotlib.rc('font', **{'family': 'IPAPGothic'})	
10	# For macOS	
11	#matplotlib.rc('font', **{'family': 'Hiragino_Maru_Gothic_Pro'})	
12	#	
13	# x の 1 乗 ~ 4 乗をプロットする	
14	#	
15	steps = 100	
16	order = 4	
17	maxx = 2	
18	#	
19	# 要素の値 0 で steps 行, order 列の行列を作成	
20	#	
21	dataList = np.zeros((steps, order))	
22	#	
23	# 凡例用のリスト	
24	#	
25	legend_label = []	
26	#	
27	# x の値を linspace で作成	
28	#	
29	x = np.linspace(0, maxx, steps)	
30	#	

```
31 # 各列について、一気に計算する
32 #
33 for j in range(1,order+1):
34     datalist[:,j-1]=x**j
35     legend_label.append(str(j) +'乗')
36 #
37 # プロット
38 #
39 plt.plot(x,datalist)
40 plt.title('xのべき乗')
41 plt.xlabel('x')
42 plt.ylabel('x**n')
43 plt.legend(legend_label)
44 plt.show()
```

## 参考文献

- [18] Wes McKinney 著, 瀬戸山ほか訳: Python によるデータ分析入門, 第 2 版, オライリージャパン (2018)

以下のサイトのチュートリアルのページなどが参考になります.

- [19] NumPy の Web サイト  
<http://www.numpy.org/index.html>
- [20] Pandas の Web サイト  
<https://pandas.pydata.org/>
- [21] Matplotlib の Web サイト  
<https://matplotlib.org/>
- [22] TkAgg のボタン操作など（見つけにくい）  
[https://matplotlib.org/users/navigation\\_toolbar.html](https://matplotlib.org/users/navigation_toolbar.html)

# 15. 振り返りとこれから

---

## 15.1 本章の学習の目標

1. この授業はここで最後です。この章ではこれまでの学習で何を学べたのか振り返ります。
2. 本授業では Python の演習を統合環境 IDLE で学びました。IDLE を採用した理由は機能が限定されていて学びやすいからですが、他にどのようなものが利用できるのかを知ります。
3. 学んだことをどう活かすのかを考えます。

## 15.2 振り返り

この授業の受講前と今とを比較して学習を振り返ってください。

- 何ができるようになりましたか
- 受講前の想定・期待とどのように違っていましたか
- これから、どのような学習目標を設定しますか

## 15.3 守破離

学期末にこの授業を受講してくれた学生の方々から意見を伺ったところ、何人の方々から「Python のプログラムを読めるようになったけれど、自分で書ける気がしない」というご意見をいただきました。実際のところ「読めるようになった」ことは大きな進歩で、プログラムを理解しながら実行してみることはできますし、少し改変してみることは簡単でしょう。これらの経験を積んでいけば、しだいに自分でプログラムを書けようになります。

武道などで「守破離」という言葉を使いますが、プログラミングも先人が書いたプログラムを読んで、実行し、すこし変えて遊んだり、応用したりしながら、知識を蓄えていけば、やがて創造的なプログラミングができるようになると思います。

## 15.4 Python の利用環境

この授業では初学者にとっての利用しやすさから構成の単純な IDLE を Python の

統合利用環境として用いました。ただ IDLE は単純なので「捨てられる」環境ともいわれています。Python については、さまざまな利用環境があります。

- Anaconda では Jupyter Notebook や Spider が含まれています。
- これらは ipython という Python シェルよりもさらに対話的な実行環境で稼働します。
- このほかにも Python に適したエディタとコマンドラインでの実行 (python, ipython) というスタイルも使われます。

## 15.5 モジュール等の追加

Python の特徴の一つが多くの方々がさまざまなライブラリを開発してくれているということです。具体的な応用例について web 上などのさまざまな情報が掲載されていますが、これらを利用する際にはライブラリであるモジュールの追加が必要になります。Anaconda では conda コマンドで、anaconda で対象としていないものや、配布パッケージとして Python を使われた方は pip コマンドを使ってモジュールの追加を行います。

## 15.6 本書で紹介しなかった話題

Python は応用の広いプログラミング言語ですが、応用に際しては応用に関連した領域の知識が必要になります。本書で紹介した NumPy や pandas も数値計算や統計処理の知識がないと使えません。このため、関連領域の知識が必要な話題は取り上げませんでした。具体的には以下のようないものが挙げられます。ご自身の興味に沿って勉強して頂ければと思います。

- ネットワークや web 関連の話題
- 画像処理などマルチメディアを扱う話題
- データベースを扱う話題
- 機械学習など人工知能関連の話題

## 15.7 感謝と恩返し—学んだことをどう活かすのか

家庭で木工をする人は少なくありません。木工ができると Do It Yourself で簡単な家具などを自分で作ることで住まいの問題解決ができるようになります。一方

で、ボール盤、旋盤、フライス盤などの機械が必要な金属を加工することは少し敷居が高いでしょう。ですので、DIY的に金属製品を作ろうとする人はあまりいません<sup>1</sup>。

Pythonによらずコンピュータのプログラミングができるようになると、「こういうことはコンピュータでやれるはず」という「ものの見方」ができるようなっています。それなら、皆さん自身がコンピュータやプログラミングを通じて社会にどう貢献するかをぜひ考えてください。

Pythonを含め、コンピュータやプログラミング言語などのソフトウェアは「誰かが作ってくれた」もので、いわば多くの技術者やプログラマーからの贈り物です。プログラミングを楽しんだら、このことに感謝し、ぜひ恩返しをしましょう。

---

<sup>1</sup> 筆者が中学生のとき、自宅が鉄工所だった友人は自分たちで自転車をつくることを考えていました。何ができる環境に居るかで自分ごととして考えられることが変わるのであります。

# 16. IDLE Python 便利帳

## 16.1 Python 便利メモ

- `help()` 関数：モジュールや関数を引数として説明を読みます。
- `globals()`：グローバルに定義されている変数などが表示されます。
- `id(x)`：オブジェクト `x` の番号が分かります。異なる変数が同じオブジェクトを指すのか確認できます。
- `type(x)`：オブジェクト `x` の型が分かります。変数にどのようなオブジェクトが代入されているかを確認できます。

## 16.2 ファイル名に注意

インポートするモジュールを作成するファイル名には使わない。

Python はモジュールを定められたフォルダで探索します。実行するファイルと同じフォルダはモジュール探索の対象です。例えば、`turtle` モジュールを使うときに、`turtle.py` という名称でファイルがあると、Python は間違って、このファイルをモジュールと理解します。

## 16.3 IDLE メモ—Python シェルのキー操作

- Ctrl-C: 実行中のプログラムの中止
- Ctrl-D: 端末入力でのファイルの終わり
  - 注意：Shell での対話モードでは shell が終了します。
- TAB キー：スマートインデント
  - 文字に続けると補完候補を表示
- ALT-P: 履歴を戻る（すでに入力した行などを再利用できます。P: previous）
- ALT-N: 履歴を次に（すでに入力した行などを再利用できます。N: next）
- スクリプトの実行
  - エディタで作成したスクリプトの実行後はその環境で対話モードとなります。スクリプト内の関数呼び出しやグローバル変数の確認が可能です。

## 16.4 IDLE メモーエディタ

### キーボード操作

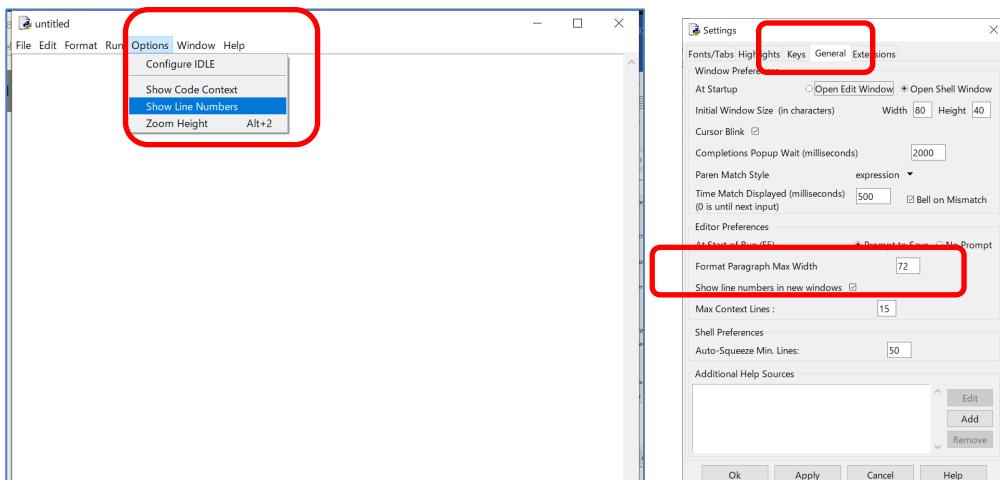
- Ctrl-]: 選択した範囲をインデント
- Ctrl-[: 選択した範囲のインデントを減らす
- ALT-3: 選択した範囲をコメント化
- ALT-4: 選択した範囲のコメント化をもどす
- Ctrl-BS: 左側の1語を消す
- Ctrl-Del: 右側の1語を消す

Windows 全般で利用可能な以下のショートカットキー操作も有用です。

- Ctrl-X: カット
- Ctrl-C: コピー
- Ctrl-V: 貼り付け

### 16.4.1 行番号表示

Python 3.8 から IDLE エディタでソースコードの横に行番号を表示できるようになりました。Options メニューで Show Line Number を選ぶか、Configure IDLE の中で、General タブを選び、Show line numbers in new windows をチェックするようにします。



また、実行時に Run ... Customized メニューを選ぶことで、コマンドライン引数の引き渡しが可能になりました。（コマンドライン引数は本書では解説していません。）

## 17. IDLE/Python でのエラーメッセージの読み方

プログラムが複雑になってくると、ただソースコードを書いて実行するだけでも、簡単なミスでさまざまなエラーが発生します。IDLE ではプログラムのエラーは

- IDLE エディターが表示するソースコードの文法上の誤り
- Python Shell が表示する実行時のエラー

の 2 種類の方法で示されます。

以下、代表的なエラーとメッセージの意味を見ておきましょう。エラーはプログラムを解釈し、実行するコンピュータが処理を継続できなくなった時点で発生します。実際にプログラマが発生させた誤りとは異なる場所であることも少なくありません。エラー表示の意味などを考えながらメッセージを読む必要があります。

### 17.1 IDLE エディタが表示するエラー

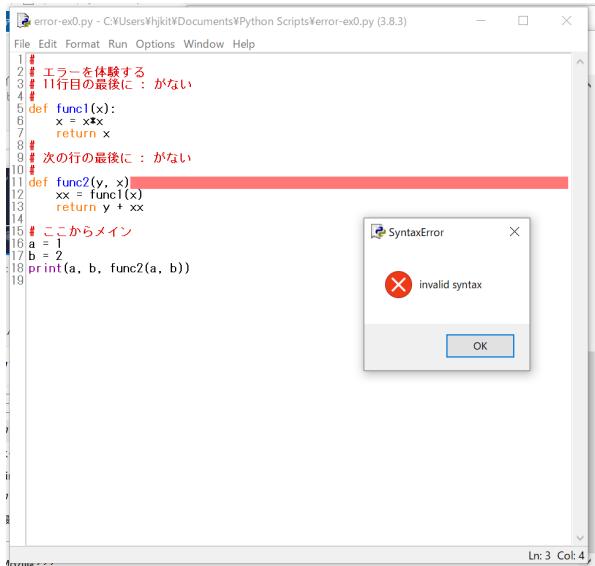
文法上の誤りは実行する前にチェックされ IDLE エディタが表示します。

#### 17.1.1 プログラム例 1 — コロン忘れ

プログラム 17-1 missing\_colon\_error.py

行	ソースコード	備考
1	#	
2	#_エラーを体験する	
3	#_11 行目の最後に _:_がない	
4	#	
5	def func1(x):	
6	_____x_=x*x	
7	_____return x	
8	#	
9	#_次の行の最後に _:_がない	
10	#	
11	def func2(y, x)	
12	_____xx_=func1(x)	
13	_____return y + xx	
14		
15	#_ここからメイン	
16	a_=1	
17	b_=2	
18	print(a, b, func2(a, b))	この最後に : が必要

この例では次の図のように「Invalid syntax (文法的に不適切)」というダイアログが示され、該当箇所が赤く表示されます。



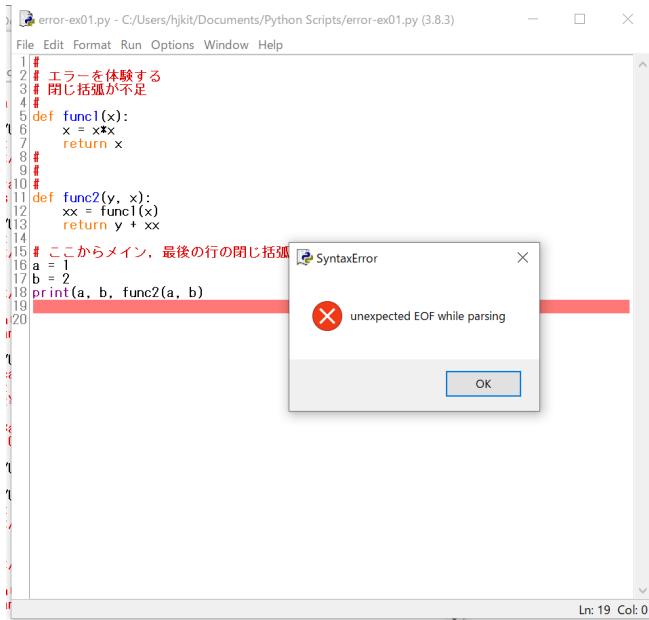
### 17.1.2 プログラム例2—括弧不足

プログラム 17-2 missing\_parentheses\_error.py

行	ソースコード	備考
1	#	
2	#_エラーを体験する	
3	#_閉じ括弧が不足	
4	#	
5	def func1(x):	
6	x=x*x	
7	return x	
8	#	
9	#_	
10	#	
11	def func2(y, x):	
12	xx=func1(x)	
13	return y+xx	
14		
15	#_ここからメイン,_ 最後の行の閉じ括弧が不足	
16	a=1	
17	b=2	
18	print(a, b, func2(a, b))	閉じ括弧が不足

この例では次の図のように「unexpected EOF while parsing (構文解析中に予期せぬファイルの終わり(EOF, end of file)になった)」というダイアログが示され、該当箇所が赤く表示されます。開き括弧と対になる閉じ括弧を探している間にソースコードが終わってしまったことを意味し、実際のエラーがある 19 行目より後ろで警告

しています。



The screenshot shows a Python script named 'error-ex01.py' in IDLE. The code contains several syntax errors, including missing colons and incorrect indentation. A 'SyntaxError' dialog box is displayed at the bottom of the screen, indicating 'unexpected EOF while parsing'. The script content is as follows:

```

1 # エラーを体験する
2 # 閉じ括弧が不足
3 #
4 def func1(x):
5     x = x*x
6     return x
7 #
8 #
9 #
10 def func2(y, x):
11     xx = func1(x)
12     return y + xx
13 #
14 #
15 # ここからメイン、最後の行の閉じ括弧
16 a = 1
17 b = 2
18 print(a, b, func2(a, b))
19 #
20

```

### 17.1.3 プログラム例 3—インデントのズレ（不足）

プログラム 17-3 insufficient\_indentation\_error.py

行	ソースコード	備考
1	#	
2	# エラーを体験する	
3	# インデントのズレ	
4	#	
5	def func1(x):	
6	x=x*x	
7	return x	
8	#	
9	# 1 3 行目の字下げがズれている	
10	#	
11	def func2(y, x):	
12	xx=func1(x)	
13	return y+xx	ここで 1 文字空白が少ない
14	#	
15	# ここからメイン	
16	a=1	
17	b=2	
18	print(a, b, func2(a, b))	

この例では 13 行目の字下げが 1 文字足りないのですが「unindent does not match outer indentation level」と表示されます。インデントを浅くしているが、同じレベルのものがない、と警告しています。

```

error-ex02.py - C:\Users\hjikit\Documents\Python Scripts\error-ex02.py (3.8.3)
File Edit Format Run Options Window Help
1 # エラーを体験する
2 # インデントのずれ
3 #
4 def func1(x):
5     x = x*x
6     return x
7 #
8 # 13行目の字下げがずれている
9 def func2(y, x):
10    xx = func1(x)
11    return y + xx
12 #
13 # ここからメイン
14 a = 1
15 b = 2
16 print(a, b, func2(a, b))
17
18

```

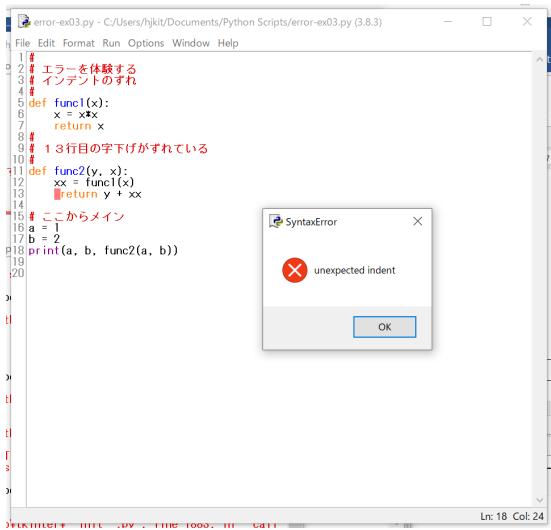
SyntaxError  
unindent does not match any outer indentation level  
OK  
Ln: 19 Col: 0

### 17.1.4 プログラム例 4—インデントのずれ（過剰）

プログラム 17-4 excess\_indentation\_error.py

行	ソースコード	備考
1	# 2 # エラーを体験する 3 # インデントのずれ 4 # 5 def func1(x): 6     x = x*x 7     return x 8 # 9 # 13行目の字下げがずれている 10 # 11 def func2(y, x): 12     xx = func1(x) 13     return y + xx 14 # ここからメイン 15 a = 1 16 b = 2 17 print(a, b, func2(a, b))	ここで 1 文字、空白が多い

今度は字下げが 1 文字多いのですが「unexpected indent」と表示されます。字下げを深くしようとしているが、それを求めるブロックがないため、予期できない字下げであると解釈しています。



## 17.2 実行時に Python Shell で表示されるエラー

### 17.2.1 プログラム例 5—未定義変数の参照

プログラム 17-5 referencing\_undefined\_variable\_error.py

行	ソースコード	備考
1	# # エラーを体験する # 6 行目で定義していない変数を参照 # 5 def func1(x): 6     xx = xx # 存在しない変数 7     return x	未定義変数 xx を参照

この例では Python Shell には以下のように表示されます。関数定義の中で生じているエラーですので、エラー箇所の呼び出しを追って表示(traceback)されます。6 行目について「`NameError: name 'xx' is not defined`」名前 xx が未定義であると表示されています。

```
>>>
Traceback (most recent call last):
```

```

File "M:\Documents\Python Scripts\error-ex1.py", line 16, in <module>
    print(a, b, func2(a, b))
File "M:\Documents\Python Scripts\error-ex1.py", line 10, in func2
    xx = func1(x)
File "M:\Documents\Python Scripts\error-ex1.py", line 6, in func1
    x = xx #存在しない変数
NameError: name 'xx' is not defined
>>>

```

## 17.2.2 プログラム例 6—引数の型違い

プログラム 17-6 wrong\_argument\_type\_error.py

行	ソースコード	備考
1	# エラーを体験する	
2	# math.sin() を呼び出す際に引数が文字列	
3	#	
4	import math	
5	def func1(x):	
6	xx=math.sin(x)	x の内容は文字列 "2"
7	return xx	
8		
9	def func2(y, x):	
10	xx=func1(x)	
11	return y+xx	
12		
13	# ここからメイン	
14	a=1	
15	b="2"	
16	print(a, b, func2(a, b))	

この例では math.sin() を呼ぶ際の引数に “2” という文字列が渡され「TypeError: must be real number, not str」と表示されています。

```

Traceback (most recent call last):
File "M:\Documents\Python Scripts\error-ex2.py", line 17, in <module>
    print(a, b, func2(a, b))
File "M:\Documents\Python Scripts\error-ex2.py", line 11, in func2
    xx = func1(x)
File "M:\Documents\Python Scripts\error-ex2.py", line 7, in func1
    xx = math.sin(x)
TypeError: must be real number, not str
>>>

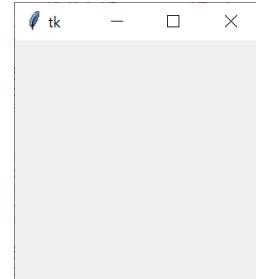
```

### 17.2.3 プログラム例 7—クラス内のインデントの間違い

プログラム 17-7 incorrect\_indentation\_in\_class\_error.py

行	ソースコード	備考
1	#	
2	#_エラーを体験する	
3	#_クラスの中の字下げ	
4	#	
5	import math	
6	import tkinter as tk	
7	class MyFrame(tk.Frame):	
8	def __init__(self, master=None):	
9	super().__init__(master)	
10	self.b = tk.Button(self, text="Try!", command=self.do)	
11	self.b.grid(row=0, column=0)	
12	#	
13	#_以下、字下げが 1 段深い	
14	#	
15	def do(self):	このメソッドが__init__メソッドの定義のなかにあることになる
16	self.b.configure(text="Did")	
17		
18		
19	root = tk.Tk()	
20	f = MyFrame(root)	
21	f.pack()	
22	tk.mainloop()	

この例は tkinter で Frame クラスを拡張子したクラス定義をつかいますが、その中でメソッド do の定義のインデントが 1 段深くなっています。このためボタンのコールバック関数として呼んだときに MyFrame に属性 do がないとされています。また、ボタンウィジェット b の生成で失敗するので右図のように tkinter のウィンドウのみが表示されます。



```
Traceback (most recent call last):
  File "M:/Documents/Python Scripts/error-ex3.py", line 20, in <module>
    f = MyFrame(root)
  File "M:/Documents/Python Scripts/error-ex3.py", line 10, in __init__
    self.b = tk.Button(self, text="Try!", command=self.do)
AttributeError: 'MyFrame' object has no attribute 'do'
>>>
```

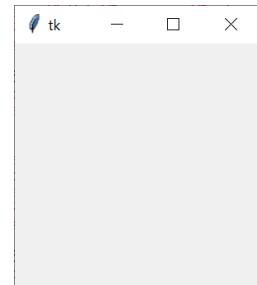
### 17.2.4

## 17.2.5 プログラム例 8—オプション引数のまちがい

プログラム 17-8 incorrect\_optional\_argument\_error.py

行	ソースコード	備考
1	#	
2	# エラーを体験する	
3	# ウィジェットのオプション間違い	
4	#	
5	import math	
6	import tkinter as tk	
7	class MyFrame(tk.Frame):	
8	def __init__(self, master=None):	
9	super().__init__(master)	
10	self.b = tk.Button(self, text="Try!", command=self.do)	command になっている
11	self.b.grid(row=0, column=0)	
12	def do(self):	
13	self.b.configure(text="Did")	
14		
15		
16	root = tk.Tk()	
17	f = MyFrame(root)	
18	f.pack()	
19	tk.mainloop()	

この例は tkinter で Frame クラスを拡張子したクラス定義をつかいますが、その中で 10 行目のボタンウィジェットのオプションの綴りがまちがっています。ボタンウィジェット b の生成で失敗するので右図のように tkinter のウィンドウのみが表示されます。エラーが発生する箇所が tkinter のモジュール内なのでわかりにくいですが、10 行目から呼び出したときのエラーであること、「`unknown option "-command"`」となっていてオプションの指定がおかしいことがわかります。



```
Traceback (most recent call last):
  File "M:/Documents/Python Scripts/error-ex4.py", line 17, in <module>
    f = MyFrame(root)
  File "M:/Documents/Python Scripts/error-ex4.py", line 10, in __init__
    self.b = tk.Button(self, text="Try!", command=self.do)
  File "M:\anaconda3\lib\tkinter\_init_.py", line 2645, in __init__
    Widget.__init__(self, master, 'button', cnf, kw)
  File "M:\anaconda3\lib\tkinter\_init_.py", line 2567, in __init__
    self.tk.call(
_tkinter.TclError: unknown option "-command"
>>>
```